DEVELOPING A SOFTWARE ENGINEER'S ENERGY-OPTIMIZATION DECISION SUPPORT FRAMEWORK

by

Irene Manotas Gutiérrez

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Fall 2017

DEVELOPING A SOFTWARE ENGINEER'S

ENERGY-OPTIMIZATION

DECISION SUPPORT FRAMEWORK

by

Irene Manotas Gutiérrez

Approved: _____
Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____
Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Lori Pollock, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

James Clause, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Benjamin A. Carterette, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

William G.J. Halfond, Ph.D.
Member of dissertation committee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Reducing the energy usage of software is becoming more important in many environments, from supercomputers to battery-powered mobile devices, and embedded systems. Recent empirical studies in green software engineering indicate that software engineers can support the goal of reducing energy usage of their applications through design and implementation decisions. However, the large number of possible design and implementation choices and the lack of feedback and information available to software engineers necessitates some form of automated decision-making support.

Two major tasks are needed to support developers in creating more energy efficient applications: (1) to understand developers' needs, and (2) to develop support tools that target the identified demands. By understanding which challenges developers face when trying to improve the energy efficiency of their applications, we can focus on appropriate approaches that target developers concerns. Focusing on developers needs, we can propose support tools that facilitate the developer's task when trying to improve applications' energy efficiency. This dissertation presents the results and implications of an empirical study about software developers' perspectives on green software engineering. The results of our study help us to understand the strategies, challenges, and needs of practitioners in industry when they attempt to improve the energy efficiency of their software applications. Furthermore, we present the design and implementation of the Software Engineers Energy Decision Support (SEEDS) framework. We show how the SEEDS framework helps developers understand the energy implications of high-level code changes, and automatically find improved version of software applications in terms of its energy consumption. Finally, we show different instances of the SEEDS framework, as well as alternative search strategies that can be used in SEEDS to guide the exploration of energy efficient solutions.

Chapter 1

INTRODUCTION

Inherent in today's computing environments are concerns about battery life, heat creation, fan noise, and overall potentially high energy costs. Research has shown that there are many ways to reduce power consumption, including designing computer architectures that are more energy efficient (e.g., [35, 40, 71, 77, 125, 145]), developing compiler optimizations that target energy usage (e.g., [64–67, 74, 78, 127, 133]), improving operating systems to help manage energy usage (e.g., [46, 119–121, 156]), and designing hardware and batteries with power consumption in mind (e.g., [3, 23, 41]).

In the Software Engineering research community, green software engineering—the process of helping practitioners (architects, developers, testers, managers, etc.) write more energy efficient applications—is increasingly targeted as an important problem area by researchers. The growing number of publications in events such as the International Workshop on Green and Sustainable Software (GREENS) [52], International Workshop on Measurement and Metrics for Green and Sustainable Software (MEGSUS) [97], and the Energy Aware Software-Engineering and Development Workshop (EASED) [44] as well as tracks at conferences such as the International Conference on Software Engineering (ICSE) [68] and the International Conference on Software Maintenance and Evolution (ICSME) [69], are examples of the growing and widespread interest in this research area.

Most existing research into helping software engineers reduce energy usage is empirically-based and has the goal of understanding how different types of changes impact the overall energy usage of applications. In particular, there have been studies of how refactorings [28, 131], design patterns [24, 91, 130], obfuscation techniques [132], applying software changes [12, 61, 70, 101], and using various implementations of an

algorithm [19] can each impact energy usage. To help developers analyze and improve the energy consumption of their applications, some researchers have proposed strategies to estimate the energy usage of applications [1, 10, 54, 58, 62, 116], and to detect energy bugs, energy leaks, and energy hotspots [92, 115, 150, 158]. Furthermore, there are a couple of works that have proposed ways to help developers improve the energy consumption of applications by using new implementations for design patterns [108], strategies to handle HTTP requests [82], and updating the color combinations used in mobile applications' GUI [90]. More recently, few works have started to propose strategies to automatically improve the energy usage of application with the help of metaheuristic optimization methods [16, 17, 90, 135].

Despite the increasing popularity of green software engineering as a research topic, little is known about practitioners' perspectives on this topic. Basic questions such as "What types of software commonly have requirements about energy usage?", or "How do developers find and correct energy usage issues?" do not have clear answers. Without understanding practitioner's needs, tools and techniques proposed by researchers and designed to make practitioners' lives easier are underused in practice (e.g., [8, 73]). Also, although the knowledge gained from empirical studies can increase our understanding of both potential energy-related "bugs", or energy efficient choices, providing only such knowledge is unlikely to be effective at reducing energy usage in practice for several reasons. First, the many layers of abstraction in typical applications, combined with subtle interactions between both hardware and software components, suggests that it is difficult, if not impossible, for developers to predict how the changes they make will impact the energy consumption of their applications. Second, the energy consumption of an application or software component can vary depending on where it is executed (i.e., hardware architecture or operating system). In practice, there are far fewer combinations that are commonly used, but there are still too many to expect developers to maintain separate versions of their applications for each possibility. Finally, it is unlikely that a single action will always result in the best outcome. In many cases, additional factors (e.g., the context of where a change

will be made) can affect the impact of a change. This means that developers need to have essentially perfect knowledge about their systems to be able to make a "good" decision.

In this dissertation, we address two major problems: The lack of knowledge about what practitioners think and do with regard to applications' energy usage during the different phases of software development, and the need of providing developers with tools that help them take decisions that improve the energy consumption of their applications. Specifically, this dissertation makes the following contributions in helping developers and researchers understand the current state of practice of green software engineering:

(1) A study that presents how the selection of web servers can impact the energy usage of web applications, (2) A study of practitioners' perspectives on green software engineering that includes interviews with 18 professional practitioners from Microsoft, and a survey of 464 developers and testers from ABB, Google, IBM, and Microsoft, (3) An analysis of the collected data that identifies practitioners' perspectives on green software engineering throughout the software development process, and (4) A discussion that both contextualizes the state-of-the-art in green software engineering research with respect to the study's findings, and suggests, for each stage of the software development process, directions for future green software engineering research.

This dissertation also presents the following contributions for enabling developers to explore the energy impacts of the decisions they make, and helping developers make their applications more energy efficient: (1) A fully automated framework, SEEDS, to support developers in the task of improving the energy usage of their applications for a given platform by making decisions about which source-level changes to apply, (2) The design of three instantiations of the framework to improve the energy usage of applications by enabling the exploration and application of different common software engineers decisions, (3) The implementation and evaluation of $\text{SEEDS}_{\text{api}}$, an instantiation of SEEDS to improve the energy usage of an application by selecting the most efficient implementations of the Collection API, (4) A case of study of how

SEEDS can be used to expand the current body of knowledge on designing and implementing energy efficient applications by enabling researchers to answer questions that they would otherwise not be able to answer, (5) The examination of two cost-reduction strategies based on the selection of energy-efficient implementations and on hotspot location sites in applications to improve $\text{SEEDS}_{\text{api}}$ costs while maintaining its effectiveness, and (6) The exploration of evolutionary optimization algorithms used as search strategies in $\text{SEEDS}_{\text{api}}$ to help with the search for energy efficient applications versions that consider multiple combinations of changes related with software developer decisions.

The remainder of this dissertation is organized as follows. Chapter 2 provides background information on related work. Chapter 3 describes two studies, one to understand how the selection of web servers can have an impact on the energy usage of web applications, and the second study about practitioners' perspectives on green software engineering. In Chapter 4 we present our novel framework, along with three instantiations, and the implementation of one instantiation, to support software developers to analyze and improve automatically the energy usage of their applications. Finally, in Chapter 5 we present a summary of our contributions and discusses potential future work.

Chapter 2

BACKGROUND AND STATE OF THE ART

With the growing usage of computers, from servers to hand-held devices, the concern about the amount of energy consumed by these machines has increased as well. Although computers are used in a wide variety of contexts, one factor remains constant: They would be more energy efficient if the software that they execute was designed and implemented with regard for energy usage. Historically, most software engineers have primarily focused on quality attributes such as correctness, performance, reliability, and maintainability while concerns about energy usage were left for compiler writers, operating system designers, and hardware engineers.

This chapter describes approaches for measuring the energy consumption of software, and mechanisms for analyzing, detecting, and improving the energy consumption of systems and software applications. First, different techniques to measure the energy usage of applications are described. Then, approaches proposed for or by system engineers to reduce or manage the energy consumption of systems are shown in the second section. The third section describes studies that present evidence upon which Software Engineers' decisions impact the energy usage of applications. Finally, the last section of this chapter highlights different approaches (e.g., techniques and tools), at the Software Engineer level, that have been proposed to examine or improve the energy efficiency of Software applications.

2.1 Measuring Energy Consumption

To improve applications' energy efficiency, it is necessary to measure the amount of energy consumed by a software application. Several approaches to measure the energy usage of software are available from works in the area of energy usage measurement.

Hardware instrumentation-based approaches (e.g., [25, 141, 151]) use physical instrumentation (i.e., soldering wires to power leads) to measure the actual power usage of a system. For instance, the Low Power Energy Aware Processing (LEAP) platform [141], provides a computer system where the amount of power consumed by each component is continuously sampled, and a program is used to synchronize power samples with portions of an application execution. Despite the high precision of hardware-instrumentation approaches, such as LEAP, they are limited in practice by software developers and researchers because they are difficult to use, and require costly and specialized hardware components.

Simulation-based approaches (e.g., [14, 55, 102] use a cycle-accurate simulator to replicate the actions of a processor at the architecture level and estimate energy consumption of each executed cycle. Like hardware instrumentation-based approaches, simulation-based approaches can be accurate, but they are only good if there exist models of the hardware and environment under test.

Finally, *estimation-based approaches* (e.g., [5, 20, 27, 38, 57, 58, 61, 84, 109, 112, 129, 136, 137, 149] can be classified into *model-based* and *software-based* approaches. *Model-based estimation approaches* build models based on energy-influencing features for a given hardware device and use such models to estimate energy usage. For example, Hao et al. and Seo et al. construct energy models of Java bytecode and then use the models to estimate the CPU energy usage of a given method or execution path within 10% of ground-truth measurements [57, 136, 137]. Also, the Opacitor approach [20] uses the energy model of the Java opcodes created by Hao et al. [58] to compute the total energy usage of Java applications. *Software-based estimation approaches* estimate

the energy usage by relying on the system's functionality of a device (e.g., CPU frequency, transmitted bytes, etc.) [110, 112, 113, 116, 129]. For instance, the Running Average Power Limit (RAPL) interface [129] offers a set of counters providing energy and power consumption information. RAPL estimates energy usage by using hardware performance counters and I/O models on certain Intel ™ microprocessors. Similarly, Jalen [111] and Jolinar [112] are other tools, based on hardware characteristics and software resource utilization, that estimate the energy usage of Java applications with minor variations between the estimated values and the actual energy consumption of applications. In general, estimation-based approaches are frequently less accurate than hardware instrumentation-based or simulation-based approaches, but they have the benefits of being easier to use and more widely applicable.

2.2 Strategies for Systems Energy Efficiency

The problem of improving the energy efficiency of systems have been addressed by researchers using several approaches at three different levels: hardware, compiler, and operating systems.

2.2.1 Hardware Approaches

Hardware energy-reduction techniques help reduce the energy consumption of hardware resources by using different strategies including the reduction of excessive CPU cycles (e.g., [143]), capping RAM energy consumption (e.g., [30]), and adding special cores to support common virtual machine (VM) operations (e.g., [21]). There is also several energy saving approaches in the area of High Performance Computing (HPC) [40, 49, 59, 99]. For instance, assigning threads to a subset of the processors to enable power-gated sleep mode for unused processors is one strategy used in HPC to reduce the energy consumption of the CPU while not degrading performance (e.g., [71, 77, 105, 125]).

Although significant energy reductions can be made by power-oriented hardware techniques, the increasingly critical power constraints and diversity of hardware

platforms and configurations have made it important to also look for software-level power optimization techniques.

2.2.2 Compiler Approaches

At the compiler level, the approaches for improving systems energy usage have focused on improving code to use fewer instructions or a more efficient ordering of instructions; controlling hibernation, implementing dynamic frequency and voltage scaling; and remote task mapping (e.g., [31, 64, 67, 74, 78, 98, 145]).

However, compiler optimization techniques consider only low-level code transformations of software and therefore do not explode the high-level transformations that can be made in the source code to optimize energy. High-level software developers could design and implement additional energy optimizations in terms of high-level source code changes.

2.2.3 Operating System Approaches

The key thought behind Operating System (OS) level approaches is to optimize energy by managing the resources used by applications and processes. Specifically, work has focused on the goals of allowing an operating system to manage energy in the same manner as other system resources (e.g., [156]), and optimizing the balance between power and performance via the automatic selection of power policies, such as dynamic voltage scaling, during application execution (e.g., [46, 121]).

However, OS energy-management polices do not exploit energy optimizations that could be done on specific applications by considering application's characteristics such as its design or implementation details.

2.3 Software Engineering and Energy Efficiency

Although leaving concerns about energy usage to the lower-level layers (i.e., hardware, compiler or operating system) has been a successful strategy, results of recent studies in Green Software Engineering [24, 91, 96, 130, 131] indicate that software

engineers could also play an important role in reducing the energy usage of applications and software systems. Software engineers could be more successful at creating energy efficient applications if they have a a better understanding of the implications of high-level design and implementation choices with regard to energy usage. The following subsections describe works that study the impacts of practitioners' decisions on applications' energy usage, and the tools and approaches that have been proposed to help practitioners be successful in improving applications' energy usage.

2.3.1 Exploring Impacts of Software Engineering Decisions

Software developers make decisions daily regarding the design and implementation of the applications they write. Usually, design decisions involve the selection of an architectural style (e.g., publish-subscribe, peer-to-peer, client-server, etc.) that defines the abstractions for representing the structure, behavior, and key properties of a software system, or that require the selection of different design patterns (e.g., abstract factory, proxy, etc.), which provide general reusable solutions to commonly occurring problems.

The energy impact associated with the selection of architectural styles and design patterns has been the subject of study in previous work [24, 91, 138], including ours, where we investigated the impacts on applications' energy usage of applying 15 design patterns from the creational, structural, and behavioral categories [130]. The results from design pattern studies indicate that the impacts on energy usage of applying design patterns vary (i.e., in some cases design patterns increase energy usage while in others, design patterns help to reduce the energy usage of applications), and are not consistent within a design pattern category (i.e., not all behavioral design patterns increase the energy usage of applications) [24, 91, 130]. Although few investigations of the energy impacts of using different architectural styles have been conducted, in [138], Seo et al. presented a theoretical framework to estimate the energy usage of the client-server, publish-subscribe, C2, peer-to-peer, and pipe- and-filter distributed architectural styles. Their results show how the selection of the appropriate architectural

style for an application can help to reduce its energy usage.

Other developers' decisions such as applying refactoring strategies have been subject of investigation as well [28, 131]. For instance, the study of Sahin et al., where the energy impacts of several refactorings techniques on Java applications is investigated, confirms how developer's decisions about whether to apply a refactoring strategy in their applications can impact their application's energy usage. Studies about how the selection of algorithms affect applications' energy consumption include the work of Bunse et al., where different sorting algorithms are compared by capturing the CPUs energy consumption when the algorithms are run on an embedded system [19]. They found that the most energy-efficient sorting algorithm is insertionsort, under the circunstances they considered.

The energy impact of software changes has been also investigated [12, 60, 61, 70, 101]. For example, Hindle studied how different software changes induce alterations in applications' energy usage by analyzing several revisions of a standalone application [60, 61], and Bhattacharya et al. investigated the energy savings resulted from Java runtime bloat reduction across different hardware configurations [12]. More recently, Imamura et al. studied the power efficiency of several code-level performance tunings for two popular High Performance Computing (HPC) programs. Their results show how different code-level changes can improve the power efficiency of programs, and that different types of code-level tunings exhibit different trends in power efficiency when CPU frequency is varied [70]. Additionally, researchers have also studied the impact of energy greedy APIs [81, 89], the impact of using different thread management constructs [123], the application of different obfuscation techniques [18, 132], and the manipulation of GUI colors [2] for mobile applications.

All these studies indicate that the decisions that developers make can have a significant impact on the energy usage of their application. Therefore, developers would benefit from help to understand how and why their decisions affect the application's energy usage and thus make better selections about the designs and implementations that are more appropriate for their applications' energy efficiency.

2.3.2 Analyzing, Detecting, and Improving the Energy Usage of Software

Although most of the research in green software engineering has focused on analyzing the impact of software engineers' decisions on the energy consumption of applications, there are some works that have focused on developing tools to help developers analyze, detect, and improve the energy usage of their applications, as well as models to support the green software development process.

To help developers analyze the energy consumption of their applications, strategies that estimate the energy usage of mobile applications at the code level [58], or for the whole application [1, 10, 54, 62, 116] have been proposed. An example of an approach to analyze applications' energy usage is GreenMiner [62], a software/hardware framework composed of a test harness, which automatically executes Android application tests while physically measuring their energy consumption. GreenMiner allows developers to obtain the energy consumption of mobile applications on certain devices, via testing and without requiring the developer to use specialized hardware. Similarly, GreenAdvisor [1], is a tool that records and analyzes an application's system calls to inform developers whether changes in an application's code have led to changes in its energy consumption.

To help developers detect issues related to the energy usage of their applications, researchers have proposed a variety of approaches, including the detection of energy bugs, energy leaks, and energy hotspots [92, 115, 150, 158]. The work of Pathak et al., presents a taxonomy of energy bugs that include energy bugs triggered by hardware, software, or external conditions. Likewise, GreenDroid [92] is a tool that diagnoses energy problems in Android applications by monitoring sensor and wake lock operations.

To improve the energy efficiency of applications, researchers have also proposed strategies to reduce the energy usage of applications by proposing new ways to implement or update portions of an application code that are related with common software engineers decisions. For instance, Noureddine and Rajan presented transformation rules to update the implementation of the Decorator and Observer design patterns to improve the energy consumption of applications [108]. Li et al. proposed a HTTP

detection and bundle strategy to automatically improve mobile applications' energy consumption by reducing the number of HTTP requests made by a mobile app [82]. Also, Linares-Vásquez et al. proposed GEMMA, a method to automatically find and update color combinations in an application's GUI that improve an application's energy usage [90]. Lately, Genetic Improvement (GI), the process of automatically improving a systems behaviour by automatically generating application code, also known as genetic programming (GP), has been proposed as a strategy to automatically improve the energy efficiency of applications [155]. GI has been used to improve systems' performance [79], and more recently to improve energy usage of systems [135], [16], [17]. In [135], the authors used a Genetic Optimization Algorithm (GOA) to automatically find program versions that use less energy than the original program by changing the assembly code of C/C++ programs. Similarly, Bruce et al. used a GI approach to improve the energy usage of three MiniSAT's downstream applications [17].

In addition to the strategies proposed to help software developers enhance the energy efficiency of applications, some researchers have also presented models to support the overall green software development process by specifying how developers and stakeholders can collaborate to produce sustainable products [6, 106]. For instance, GREENSOFT [106] is a conceptual model that defines a software life cycle, sustainability criteria and metrics for software products, procedure models for different stakeholders, and tools that support developing, supplying, and using software in a green sustainable manner.

In general, and despite the increasing interest of the green software engineering community in developing approaches to help developers in their task of improving applications' energy efficiency, there is not a clear understanding about software engineers practices associated with green software engineering activities. Questions such as "How energy related requirements compare to other non functional requirements?", or "How software developers consider to make their applications energy efficient?" do not have a clear answer. Having knowledge about practitioners' state of practice in green software engineering can help to develop tools and techniques that support their

activities and are not underused in practice. Furthermore, most of the approaches proposed to help developers in their task of improving applications' energy efficiency are not general enough to support different software engineers decisions i.e., they are tight to a specific change (e.g., applying a refactoring, implementing a design pattern, using an API), and only a few of the proposed approaches, such as the ones based on the GI strategy, provide automatic means that developers can use to improve the energy usage of applications. Hence, to help understand the current state of practice of green software engineering by practitioners, we studied practitioners' practices and challenges when trying to design or improve application's energy efficiency and present the results in Chapter 3. Also, motivated by the results from our studies, along with the state of the art in green software engineering, we propose a software framework that helps developers to both analyze and improve automatically applications' energy usage through the application of source code transformations related to software developers' decisions; more details about our framework are presented in Chapter 4.

Chapter 3

UNDERSTANDING SOFTWARE DEVELOPERS' CHALLENGES AND NEEDS THROUGH STUDIES

This chapter first reports on a case study that investigates how the selection of a web server can impact the energy usage of web applications. Results from our case study, and others' empirical studies described in Chapter 2 motivated our study of practitioners' perspectives on green software engineering, which is described in the second part of this chapter.

3.1 A Case Study: Impacts of Web Servers on Web Applications Energy Usage

Often, when choosing what software to run in a data center, companies face a difficult choice: how to choose the most appropriate software from a set of implementations that provide similar functionality. For example, which web server should be used to execute a web application? Such decisions are often made by analyzing various aspects of the candidate implementations, such as their size, performance, stability, etc. Unfortunately, this choice becomes even more difficult when the desire to minimize energy consumption is introduced; companies lack the information needed to identify an appropriate balance between such aspects and the energy consumption of the implementations. For example, they may want to choose a web server that meets their performance needs but also consumes the least amount of energy possible.

The advent of technologies like cloud computing and Internet of Things (IoT), have help to increase the demand for web applications and web servers. Web applications popularity have increased as well thanks to factors, such as the growing number of frameworks that facilitate web applications' development, and due to the improved

capability of web browsers that enable web applications to run on multiple devices, such as desktops, laptops, and handset devices. However, when developers have to choose a web server to deploy their web applications, they can select from various web servers alternatives. Web servers are one of the most commonly used types of software in data centers. There are many different web server implementations, which suggests that there are many design and implementation decisions that may impact a specific implementation's energy consumption. Thus, in our ongoing effort to support developers and companies in making energy-efficient development choices, we investigated whether using different web servers impacts the total energy consumption of a web application. We conducted an empirical study of four web servers. At a high level, the focus of the study was two-fold: first we wanted to learn whether the choice of web server potentially alters the overall energy usage of the web applications that it serves and second, if so, how such differences can be characterized.

3.1.1 Methodology

This section describes the details of our experimental setup including our considered variables, subject applications, and data collection protocol.

3.1.1.1 Experimental Variables

In this study, we considered one dependent variable, the amount of energy consumed by a web application, and one independent variable, the choice of web server. To isolate the impacts of switching web servers (independent variable) on web application energy usage (dependent variable), it is necessary to control the impact of several extraneous variables (e.g., changes in the considered web application's code, the web browser used to interact with the web server, the user actions that are used to drive the web application). The next subsections describe how we control for these factors.

3.1.1.2 Controlling for Changes in the Web Application's Code

To fully explore the impact of different web servers, it is important to be able to switch web servers without needing to modify the web application's code. In many

situations, such changes are necessary because web servers expose different Application Programming Interfaces (APIs) for clients to interact with. Unfortunately, making such changes would introduce a potential source of bias in our experiments for several reasons. The primary reason is that we would need to make such changes ourselves. Although we are familiar with web application development, it is not our profession and as such, it is likely that the changes we make would be inefficient, subtlety incorrect, or otherwise suboptimal compared to how professional developers would modify the application.

To avoid this bias, it is necessary to be able to switch web servers without needing to modify the web application. The Rack web server interface[1] provides this ability. At a high-level, Rack is a minimal API for connecting web servers and web application frameworks. Its most common use is with Ruby on Rails (RoR) web applications. With Rack, web application developers can avoid writing handlers for every possible web server while keeping their application's handling of HTTP requests and responses simple. A Rack application has two main parts:

(1) A collection of methods that implement its functionality. Each method takes one argument, an environment that encapsulates a request, and returns an Array that encapsulates the status, headers, and body of the application's response.

(2) A set of mappings that route URLs to application methods (i.e., determine which method should be used to fulfill a given request).

Figure 3.1 shows a simple "Hello world" Rack application. The top half of the figure shows the web application's code. In this case, there is one method that implements the application's functionality (i.e., saying "hello"). The bottom half of the figure shows how simple it is to run Rack applications on a variety of web servers. Simply switching the application's handler is sufficient; no changes need to be made to the code of the application.

[1] http://rack.github.com

```
require 'rubygems'
require 'rack'

def application(env)
    [200, {"Content-Type" => "text/html"}, "Hello Rack!"]
end

---

Rack::Handler::Mongrel.run method(:application), :Port => 9292
# Rack::Handler::Puma.run method(:application), :Port => 9292
# Rack::Handler::Thin.run method(:application), :Port => 9292
# Rack::Handler::WEBrick.run method(:application), :Port => 9292
```

Figure 3.1: Example of a Rack application.

By considering web applications and web servers that support the Rack API, we gain the ability to control for the biases that would result from modifying the web application's code and can more fully assess the impact of a web server on the energy consumption of a web application.

3.1.1.3 Controlling for Inconsistencies in Driving the Web Application

In general, web applications are interactive and event-driven. They accept user input, perform some computation, and generate a response. In our experiments, this interactive, event-driven nature can introduce a potential source of bias. Unlike with batch applications, it is difficult to manually reproduce a given execution exactly. For example, a web application user can often repeatedly perform the same sequence of actions (e.g., enter text into a web form, click a button), but cannot maintain the same timing between the actions. Although such differences may seem inconsequential, they may lead to observed differences in energy consumption that are not due to the web server, but rather differences in how the web application is driven. To prevent such bias, it is necessary to be able to deterministically reproduce a given sequence of user actions with great fidelity; integration and acceptance testing frameworks provide this capability.

Integration and acceptance testing frameworks (e.g., Cucumber[2] and Capybara[3]) are commonly used as part of Test Driven Development (TDD) and Behavior Driven Development (BDD). They allow developers to describe how software should behave in a plain text format. Such behavioral descriptions are then automatically transformed into a sequence of low-level steps that are automatically carried out against a target web application.

Figure 3.2 shows an example integration test written using the Cucumber framework. *Scenarios* are the main components of Cucumber automated tests. A scenario represents a user-level acceptance test. Scenarios are described in files called *Features* and are run against the system under test. Each Scenario has a list of *Steps* that are executed when a scenario is run. Steps describe the context in which the scenario will be executed, the key action the user performs, and the expected outcome.

Because the testing framework is performing the actions instead of a user, the variability in the amount of time that lapses between performing actions is much less. Hence, any observed variations in energy consumption are more likely to be the result of changing the web server rather than inconsistencies in the user interaction.

3.1.1.4 Controlling the Web Browser

Similar to the web application's code and how the web application is driven, we need to control which web browser is used to interact with the web application as the web browser also contributes to the overall energy consumption of the web application.

Controlling for the impact of the web browser used to interact with the web server is straightforward: simply use the same web browser. In our experiments, we arbitrarily chose to use the Firefox web browser.[4]

---

[2] http://cukes.info

[3] http://jnicklas.github.com/capybara/

[4] http://www.mozilla.org/en-US/firefox/new/

```
Feature: Show all due actions in a calendar view
  As a Tracks user
  In order to keep overview of my due todos
  I want to manage due todos in a calendar view

  Background:
    Given the following user record
      | login    | password | is_admin |
      | testuser | secret   | false    |
    And I have logged in as "testuser" with password "secret"
    And I have a context called "@calendar"

  @javascript
  Scenario: Setting due date of a todo will show it in the
      calendar
    When I submit a new action with description "something new"
      in the context "@calendar"
    And I go to the calendar page
    Then the badge should show 0
    And I should not see the todo "something new"
    When I go to the home page
    Then I should see the todo "something new"
    When I edit the due date of "something new" to tomorrow
    And I go to the calendar page
    Then I should see the todo "something new"
    And the badge should show 1
```

Figure 3.2: Example of a Cucumber integration test.

3.1.1.5 Considered Subjects

For this case study, we focused on one web application and four web servers. When choosing the subject web application and web servers, we needed to ensure that they satisfied the requirements described in Section 3.1.1.1. More specifically, (1) both the application and the web servers should implement the Rack API, and (2) the application should come with a comprehensive suite of automated integration tests. Note that in addition to helping control for potential biases in how the application is executed, choosing an application with integration tests has the benefit of allowing us to measure energy consumption under expected usage scenarios.

Subject Web Application

As our subject application, we selected Tracks.[5] Tracks is a web application that is designed to help users implement David Allen's "Getting Things Done"™ methodology. Tables 3.1 and 3.2 presents some statistics for Tracks calculated using its Rake

[5] http://getontracks.org

Table 3.1: Metrics for Tracks

Component	LoC	# Classes	# Methods
Controllers	3,645	20	296
Helpers	863	0	103
Models	1,349	14	181
Libraries	513	7	79
APIs	17	1	0
Total	6,387	42	659

Table 3.2: Integration test cases

Feature	# Scenarios	# Steps
Calendar	5	47
Context edit	9	111
Preferences	4	31
Review	7	47
Search	4	53
Show statistics	7	69
Toggle context	2	38
Total	38	396

stats task.[6] Table 3.1 shows measures that indicate the size of the Tracks application. The first column in the table, *Component*, shows the names of different components in Tracks, and the subsequent three columns, *LoC*, *# Classes*, and *# Methods*, show the number of lines of code, number of classes, and number of methods, respectively, of each component. For example, Tracks' controllers contain 3,645 lines of code (LoC) divided across 296 methods in 20 classes. Table 3.2 shows statistics about the Cucumber integration tests provided with Tracks. In the table, the first column, *Feature*, shows the different features that are being tested. The remaining columns, *# Scenarios* and *# Steps*, show the number of scenarios and number of steps that are used to test each

[6] Rake is a commonly used build system similar to make or Ant.

feature, respectively. For example, the Calendar feature is tested using 47 separate steps grouped into 5 scenarios.

Subject Web Servers

As our subject web servers, we selected the following four web servers based on their popularity and their support for the Rack web service interface:

- **Mongrel:** Mongrel[7] is an HTTP library and web server. This Rack server uses a Ragel extension to provide fast, accurate HTTP 1.1 protocol parsing.

- **Puma:** Puma[8] is a thread-based server that provides a fast and concurrent HTTP 1.1 server for Ruby web applications. Puma, which is based on Mongrel, includes a small memory requirement and is considered high-speed compared to other Rack-based servers. Puma runs on all Ruby implementations.

- **Thin:** Thin[9] is a Ruby HTTP web server that supports a Mongrel parser, the root of Mongrel speed, security Event Machine, a network I/O library, and the Rack library. Thin has the highest Ruby Toolbox popularity rating between all the available Rack Ruby servers.

- **WEBrick:** WEBrick[10] is an HTTP server toolkit that can be configured as an HTTPS server, a proxy server, and a virtual-host server. WEBrick features complete logging of both server operations and HTTP accesses. WEBrick supports both basic and digest authentication in addition to algorithms not in RFC 2617. A WEBrick server can be composed of multiple WEBrick servers or servlets to provide different behaviors on a per-host or per-path basis. WEBrick includes

[7] http://mongrel2.org

[8] http://puma.io

[9] http://code.macournoyer.com/thin/

[10] http://www.ruby-doc.org/stdlib-1.9.3/libdoc/webrick/rdoc/WEBrick.html

servlets for handling CGI scripts, ERb pages, Ruby blocks and directory listings. WEBrick also includes tools for daemonizing a process, starting a process at a higher privilege level and dropping permissions. WEBRick is considered a non-production server.

Each of these web servers supports many configuration options that have the potential to impact their energy consumption. Ideally, each server would be custom configured to consume the least amount of energy. However, for our experiments, we chose to use the default configuration for each server. We made this choice for several reasons. First, we need a standard configuration for the experiments. If we were to manually configure the servers, it is possible that we would introduce a source of potential bias by creating incomparable configurations. Second, default configurations are often used in production, thus they mimic normal use. Finally, because each server is implemented differently, they do not all support the same options. For example, Thin does not have a maximum number of simultaneous connections configuration option because it runs a single event loop instead of using multiple threads. In short, creating an identical configuration for each server is impossible.

Table 3.3 presents a summary of the configuration options used for each web server. In the table, the first column, *Parameter* shows the name of the considered configuration parameter and the remaining columns, *Mongrel*, *Puma*, *Thin*, and *WEBrick*, show the value of the parameter for each of the web servers.

3.1.1.6 Data Collection

To measure the amount of energy consumed by our subjects, we ran them on a LEAP-based system [141]. Our LEAP system is an x86 platform based on an Intel Atom motherboard (D945GCLF2), configured with 1 GB of DDR2 RAM, a 320 GB 7200 RPM SATA disk drive (WD3200BEKT), and runs OpenSuse 12.1. Each component in the LEAP system (e.g., CPU, disk drives, memory, etc.) is connected to an analog-to-digital data acquisition (DAQ) card (National Instruments USB-6215) that samples the amount of power consumed by the component at a rate of 10 kHz. We

Table 3.3: Server Configuration Parameters

Parameter	Mongrel	Puma	Thin	WEBrick
Timeout (s)	60	N/A	30	30
Max. # Conn.	960	N/A	1024	100
Max. # Persist. Conn.	N/A	N/A	100	N/A
Daemonized	No	No	No	No
# Instances	1	1	1	1
Logging enabled	Yes	Yes	Yes	Yes
Threaded	Yes	Yes	N/A	Yes

selected the LEAP system to profile the energy usage of our subjects because of its accuracy and its capability to measure the energy consumption for the CPU, memory, and disk, while running each of the subjects in our study.

As an application of interest is executing, the LEAP system collects two types of information. It records the power samples gathered by the DAQ, and it uses a custom kernel module to record timing information (e.g., the CPU timestamp counter) that is necessary for synchronizing the power samples with specific points of interest in the execution. Typical points of interest are the starting and stopping times of the application, but the system also exposes an API that developers can use to indicate that arbitrary portions of an execution are interesting (e.g., method entry or exit, phase changes).

After the execution is complete, the energy consumption of each component and of the total application is computed and reported in Joules (J). Note that by using the synchronization information, the system can also report finer-grained energy usage numbers (e.g., the energy consumed by an individual method invocation or section of code).

3.1.2 Threats to Validity

One possible threat in our study is the representativeness of the subjects, i.e., the four web servers that implement the Rack library. We address this threat by

choosing web servers that are popular and publicly available. Also, the configuration of each server can bias the results; to avoid this, we chose the default configuration, as a standard configuration and one that is the most used. Furthermore, the energy measurement system used in the experiment could be considered a threat to validity. To minimize this threat, we used the LEAP monitoring system that provides high accuracy when sampling the energy data, has been used by others, and it is able to measure the energy of several components (e.g., disk, RAM and CPU), and the direct energy of discrete events in kernel and user space systems. Also, the selection of the web application can be considered another threat to validity. To minimize this threat, we selected Tracks, a popular ruby on rails web application that is open source, provides a diverse set of features to test the application, and supports the Rack API used to seamlessly switch between web servers. Another possible threat is due to the tests that serve as workloads; we cannot guarantee that we can get the same results if we run another set of tests. To mitigate this threat, we did not develop our own tests for the web application, but instead we used the integration tests that come with the web application.

3.1.3 Evaluation

We refined our overall question of whether or not the choice of web server impacts the energy consumption of a web application into the following specific research questions:

- **RQ1—Feasibility:** Does the choice of a web server impact the energy consumption of a web application?
- **RQ2—Consistency:** Are the web servers consistent in their impact across the features of a web application?

The remainder of this Section 3.1.3 presents and discusses the results in terms of these research questions.

Table 3.4: Energy consumption of Tracks when using each web server.

	Energy consumption (J)				
Feature	Mongrel	Puma	Thin	WEBrick	Mean
Calendar	1,788	1,515	1,482	1,646	1,608
Context edit	2,436	2,418	2,469	2,556	2,470
Preference	352	401	352	359	366
Review	970	922	968	1,062	981
Search	1,371	1,405	1,272	1,339	1,346
Show stats.	1,189	1,233	1,016	1,192	1,157
Toggle context	784	847	870	730	808
Total	8,890	8,741	8,429	8,884	8,736

RQ1—Feasibility

The purpose of our first research question was to investigate whether the choice of a web server can have an impact on the overall energy consumption of a particular web application. To answer this question, we ran each of Tracks' integration tests (see Section 3.1.1.5) against Tracks when it was being run by each server. As the tests were executing, we used the LEAP system (see Section 3.1.1.6) to record the energy usage of the application.

Table 3.4 shows a view of the experimental data that we gathered. The first column, *Features*, shows the name of each feature-level integration test (see Table 3.2). The next four columns, *Mongrel*, *Puma*, *Thin*, and *WEBrick*, show the actual measure of the energy usage in Joules (J) for a single run of Tracks that is consumed by each feature when the application is run on each web server. The final column, *Mean*, shows the mean of the energy consumption of the four web servers, also in Joules. Finally, the last row in the table shows the total for each column. For example, when using Mongrel, Puma, Thin, and WEBrick, running the Calendar integration test causes Tracks to consume 1,788 J, 1,515 J, 1,482 J, and 1,646 J, respectively; the mean energy consumption of the Search feature is 1,346 J; and the total energy consumption of

Tracks, across all features, when using Thin is 8,429 J.

This data indicates that the energy consumption of a web application can indeed vary depending on which web server is used. For Tracks' integration tests, Thin is the most energy-efficient web server, while Mongrel is the least efficient, consuming 6 J more than WEBrick. The small difference in the amount of energy consumed when using Mongrel and WEBrick is surprising as Mongrel is one of the most commonly used production web servers for RoR applications, while WEBrick is intended to be used only for development. The closeness in energy usage between the Mongrel and WEBrick web servers is encouraging for our research for several reasons. First, it can suggest that *design choices made to support the requirements necessary for a high-performance production server (e.g., high throughput and responsiveness) may be detrimental with respect to energy usage.* Second, it provides initial evidence for our assumption that, *to be energy-efficient, web servers might need to be selected specifically to meet the needs of a web application.*

RQ2—Consistency

The purpose of our second research question was to investigate whether the impact of a web server is consistent across features. To answer this question, we first investigated, for each feature-level integration test, how the energy consumption of each web server compares to the mean energy consumption of all web servers.

Table 3.5 shows the results of this comparison. In the table, the first column, *Feature*, shows the name of each feature-level integration test and the remaining columns, *Mongrel*, *Puma*, *Thin*, and *WEBrick*, show the percentage difference in energy consumption of each web application compared to the mean energy consumption of all web servers shown in Table 3.4. More specifically, we use equation 3.1, to compute the values in Table 3.5, where EC_{ws} represents the energy consumption of a web server, and MEC_{aws} represents the mean energy consumption for all web servers.

$$\frac{EC_{ws} - MEC_{aws}}{EC_{ws}} \tag{3.1}$$

Table 3.5: Percentage change of energy consumption from the mean.

	% Difference from the mean			
Feature	Mongrel	Puma	Thin	WEBrick
Calendar	10.1	−6.1	−8.5	2.3
Context edit	−1.4	−2.1	−0.1	3.4
Preference	−4.0	8.7	−4.0	−1.8
Review	−1.1	−6.3	−1.3	7.7
Search	1.8	4.1	5.9	−0.6
Show statistics	2.7	6.1	−13.9	2.9
Toggle context	−3.0	4.7	7.2	−10.7
Total	1.7	0.1	−3.6	1.7

As the table shows, there are differences in which feature is the most energy costly on each web server. The results indicate, for example, that on Mongrel the Calendar feature is the most expensive to execute, on WEBrick the Review feature is the most expensive, on Puma the Show statistics feature is the most expensive feature, and on Thin the Toggle context is the most expensive feature. Similarly, for each of the rows in Table 3.5, we see that the Review feature runs the most energy efficiently on Puma, Show statistics is the most energy efficient on Thin, and Toggle context is the most energy efficient on WEBrick. The results also show that on Mongrel no feature runs the most energy efficiently. The magnitude of the difference varies greatly across web servers. In some cases, there is clearly a most efficient server, such as the Thin server when the Show statistics feature is executing, whereas Mongrel and the Thin server get very similar energy costs when executing the Preference feature (the difference is less than a 1 J).

In addition to looking at the aggregated energy consumption of each feature as shown in Tables 3.4 and 3.5, we also examined the results in more detail. To that end, we studied the energy consumption of the individual scenarios in three of the feature-level integration tests, Calendar, Context edit, and Search. Figure 3.3 shows the results of this examination. This figure shows a scatterplot of the amount of energy consumed

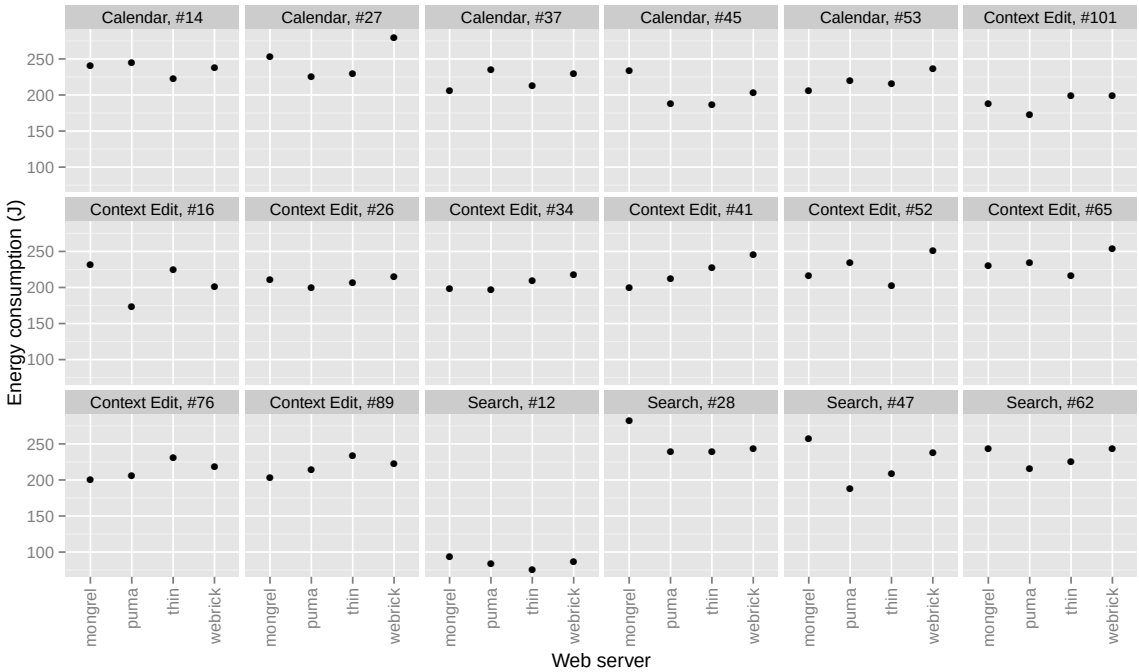

Figure 3.3: Energy usage per test and per web server for the Calendar, Context edit, and Search integration tests.

by each web server faceted by feature-level integration test and scenario identifier. Note that the scenario ids are not sequential. This is because Cucumber identifies scenarios within a feature by the line number where the scenario starts. Within each facet of the scatterplot, the x-axis shows web servers and the y-axis shows the energy consumption of the scenarios. For example, running Scenario #16 of the Context edit feature consumes ≈230 J, ≈175 J, ≈225 J, and ≈200 J when run using Mongrel, Puma, Thin, and WEBrick, respectively.

Again, we can see that in some cases, there is clearly a most efficient server (e.g., for Scenarios #101 and #16, Puma is clearly the most energy efficient; for #52 Thin is most efficient; for #53 and #37, Mongrel is most efficient). In other cases, the energy consumption of the more efficient servers is very close (e.g., for Scenario #45, Puma and Thin are within a few Joules).

Overall, the results for this research question show that the *impact of a web*

server is not consistent across the web application's features. For each web server, not only are there cases where the magnitude of their impact changes, but the choice of a web server also can both increase and decrease the overall energy usage of the application depending on which features are executed. One possible reason can be the way web servers are designed to manage the computational resources e.g., memory, while carrying out a client request related with a scenario's action step. Also, it is possible that some web servers' features are more favorable for certain steps types in an scenario. For instance, although the Thin server is based on some of Mongrel's characteristics, Thin includes a network I/O library specifically to improve its performance and scalability. This characteristic of the Thin server seems to perform well for the steps included in the scenarios of the Calendar feature, but it could be the reason why it does not perform very well in terms of its energy consumption when executing the steps included in the scenarios of the Search feature.

3.1.4 Summary of Results

In this section, we described a case study that investigated the potential impacts on Web Applications' energy usage due to the selection of different Web Servers. We considered four Web Servers complying with the Rack web server interface to serve a single application, and examined the energy consumption for the execution of seven application's feature s.

The results of this study suggest that:

(1) The energy consumption of a web application can vary up to 14% depending on the web server used to handle its requests.

(2) The impact of a web server is not consistent across web application's features.

(3) A web application's energy usage can increase or decrease depending on which of its features are executed.

3.1.5 Implications

The results presented in this study suggest that developers should carefully analyze which web server to choose when considering the energy consumption of their web applications. Researchers can use the results of the presented study to further analyze the causes that make a web server more or less energy efficient when running a web application. For instance, researchers can investigate whether the types of operations executed by a web application' scenario have the same energy consumption when using different web servers. Also, researchers can investigate whether the design choices made for a web server (e.g., frameworks or APIs being used), have an impact on the energy consumption of web applications being served.

3.2 Investigating Practitioners' Perspectives on Green Software Engineering

The research community has not been blind to the increasing concern about applications' energy usage due to dramatic shift in the type of computing devices used by consumers and enterprises e.g., millions of sales in personal computers (PCs) and tablets, compared to billions of sales in smart phones. As a result, green software engineering is increasingly targeted as an important problem area by software engineering researchers. The growing number of publications in events such as GREENS [52], MEGSUS [97], and EASED [44] as well as tracks at conferences such as ICSE [68], and ICSME [69], are examples of the growing and widespread interest in this research area.

Despite its increasing popularity as a research topic, little is known about practitioners' perspectives on green software engineering. Even basic questions such as "What types of software commonly have requirements about energy usage?", "How does the importance of reducing energy usage compare to other requirements?", and "How do developers find and correct energy usage issues?" do not have clear answers. Without understanding practitioner's needs, researchers may find themselves in a situation where, despite the investment of significant amounts of time and effort, tools

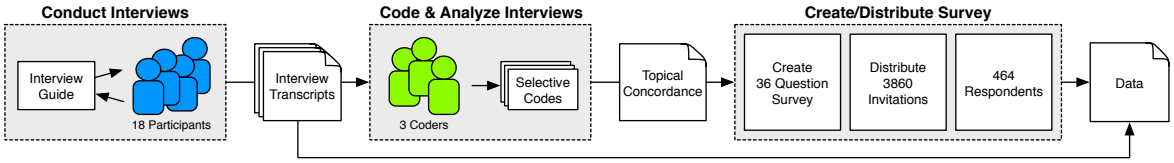

Figure 3.4: The applied research methodology.

and techniques designed to make practitioners' lives easier are underused in practice (e.g., [8, 73]).

To help inform research in green software engineering, we conducted both in-depth interviews of 18 Microsoft practitioners from a wide range of application domains and a quantitative, targeted survey of 464 ABB, Google, IBM, and Microsoft developers and testers. To the best of our knowledge, the interviews and survey compose the first broad-based empirical study of practitioners' perspectives on green software engineering —how they think about battery life/energy usage when they write requirements, design, construct, test, and maintain their software.

3.2.1 Methodology

Figure 3.4 depicts the research methodology we used for this study. At a high-level, it has two main components: interviews and a survey. Individually, each of these approaches has strengths and limitations; combining them leverages their individual strengths and reduces their individual weaknesses. Interviews are useful for gathering a wide range of qualitative observations and insights and for gaining an understanding of the broad context and environment that the interviewees operate in. In addition, their interactive nature allows for collecting in-depth information about participants' thoughts and opinions. However, their high costs restrict the number that can be performed. Conversely, surveys allow for collecting only a limited amount of data from each respondent. However, their low costs allow for reaching a large number of respondents, which provides generalizability. Conducting a survey after performing

interviews enables us to quantify and generalize the results obtained from the interviews over a larger population and to quantitatively assess themes that were implied by the interview participants.

3.2.1.1 Interviews

The first step in our methodology was to interview practitioners at Microsoft. These interviews were purely exploratory and were not intended to provide generalizability. Rather the goal of this step was to learn about how the participants think about energy usage in the context of software development from a variety of perspectives and domains.

Interviews' Protocol

We used semi-structured, in-depth interviews based on an interview guide to enable a detailed exploration of the participants' views and experiences using a flexible and responsive approach [63]. Interviews were audio-recorded at each participant's office, with participant permission, and lasted between 30 to 60 minutes each.

At a high level, the interviews had four main parts. First, the participant was asked some general demographic questions to gather information such as the interviewee's education, technical role, years of experience, and team. Second, the interviewers asked about the participant's views on energy usage of software. This positioned the participant on the spectrum of energy usage and allowed the participant to speak openly about their experiences and opinions about energy usage while limiting bias from the interviewers. Third, the interviewers began to converse with the participant by asking open-ended and clarification questions based on the second part of the interview. The interactive nature of the conversations allowed the interviewers to gather detailed information about the participant's experiences with techniques, policies, specifications, patterns, contexts, failed and successful attempts, etc. For example, questions like "Do you have a baseline platform that you use?" and "What have you seen teams do today to determine if there are energy issues with their applications?" were posed to several

participants. Finally, the interviewers thanked the participant, explained how their responses would be used, and asked whether there was anything else they wanted to mention that was not previously covered. A list of questions asked during the interview sessions can be found in Appendix A.

Interview Participants

We identified an initial group of practitioners through multiple means including using mailing lists related to energy use, querying the employee database with energy-related keywords, and communicating with product group managers to find employees that deal with energy. We included practitioners who appeared to have experience with green software engineering from areas such as mobile application development and server-side infrastructure. Because our goal was to learn about as many perspectives as possible, we ensured that the participants came from a range of projects and platforms and had various roles and levels of seniority. Such a selection strategy is called *Maximum Variation Sampling* [117] and is appropriate, as in this case, when a sample may be limited and "the goal is not to build a random and generalizable sample, but rather to try to represent a range of experiences related to what one is studying."

The initial group of participants was expanded using the *snowball process*—participants were added based on recommendations from current participants—until the *data saturation point* was reached [9]. That is, once new interviews yield no additional information, further interviews will yield only marginal (if any) value [53]. Using the snowball process allowed us to access the hidden population of experienced green software practitioners—practitioners who we would otherwise be unable to identify—without incurring prohibitive costs. In total, we interviewed 18 participants, a number similar to those used in related work (e.g., [73, 80]).

Data Analysis

We used open, axial, and selective coding to qualitatively analyze the data obtained from the interviews [50, 144]. A professional transcription service transcribed the audio recordings and divided them into 355 segments based on distinct conversation topics. We used open coding to summarize each segment. Then we used axial coding to establish relationships among the summaries. Finally, we used selective coding to identify core ideas that were expressed throughout the interviews. We defined two categories of selective codes: the first contains codes that roughly correspond to Chapters 2–6 in the Software Engineering Body of Knowledge (SWEBOK) [13], and the second contains codes that indicate the type of information provided by the participants (e.g., goals, opinions). Finally, we coded each segment using the selective codes. Because of the large number of segments, we required that each segment was assigned at least one code from each category. If none of the codes from a category was appropriate, a special "no code" code was assigned. Forcing the assignment of at least one code eliminated the possibility that coders accidentally skipped a segment. As a result of the coding process, we created a topical concordance that shows, for each code, the relevant portions of the transcripts.

3.2.1.2 Survey

The second step in our methodology was to survey practitioners. While the interviews were exploratory, the goal of this step was to quantitatively assess, over a large and representative population, the qualitative information that we learned from the interviewees.

Survey's Protocol

We used Kitchenham and Pfleeger's guidelines for personal opinion surveys [76] and the results from coding the interviews to write 155 candidate statements. Each statement asks the survey respondent to either (1) rate their agreement with the statement on a 5-point Likert scale from Strongly Disagree to Strongly Agree, or (2) indicate,

on a 5-point Likert scale from Never to Almost Always, how frequently the event described by the statement occurs. We then condensed the initial set of statements by removing statements that were redundant, ambiguous, or difficult for respondents to self-assess while ensuring that statements derived from each selective code were represented in the final list. This process reduced the set of candidate statements to 36 final statements, which we believed would keep the survey under our target of 15 minutes. Abbreviated versions of the final 36 statements are shown in Figures 3.5 to 3.10 in Section 3.2.2, and a complete copy is available online[11] and in Appendix A.

In addition, we asked free response follow up questions for areas of particular interest when the response to a question indicated the respondent may have insight or evidence to share. For example, if a respondent indicated that their applications have energy usage goals or requirements, we asked them to provide an example.

Survey Participants

We primarily recruited developers and testers as survey respondents because the statements we created reflected practices and concerns related to their work. We recruited developers and testers from all four companies who worked on applications for either mobile devices, data centers, embedded systems, or traditional PCs. To reach these groups, we selected employees based on their position in their company's organizational chart. In total, we sent invitations to 3,860 employees, 700 from ABB, 1,500 from Google, 160 from IBM, and 1,500 from Microsoft. We selected these companies because of their extensive experience on creating software, their large number of software practitioners, and because their diverse set of software products targeting different devices. The survey was anonymous, though we did ask respondents to provide their contact information if they were willing to let us follow up with them. At Microsoft, we offered a drawing for two $50 gift cards as this has been shown to improve participation at Microsoft in the past [142]. The overall response rate is 12 % (464 responses) with per-company rates of 9 % for ABB (62 responses), 9 % for Google

[11] https://surveys.research.microsoft.com/s3/DeveloperEnergySurvey

(134 responses), 13 % for IBM (21 responses), and 16 % for Microsoft (247 responses). Other online surveys in software engineering research have reported similar response rates [126].

3.2.1.3 Threats to Validity

In our interviews and survey, we avoided practitioners with no interest in energy. Thus, we may be overestimating the importance of the area as a whole. However, we were not interested in contrasting experienced and inexperienced practitioners; instead we preferred to gain insights from experienced green software engineering practitioners.

Due to the costs of interviewing practitioners, we contacted potential interviewees and provided them with a brief outline of the goals of our study. Knowing the high-level goals of the study allowed practitioners to assess whether they could provide useful information. However, because they were aware of the goals of the study, they may have provided information based on what they thought we wanted to know (hypothesis guessing) or they may have withheld information or opinions that they thought would be unpopular (evaluation apprehension) [153]. We reduced these threats by guiding the interview process and assuring the participants that their answers would be anonymized.

The fact that one of the interviewers was not a Microsoft employee may have led to participants withholding information. We addressed this threat by ensuring that at least one interviewer was employed by Microsoft and clearly stating that all relevant non-disclosure agreements had been signed.

Our interview participants were partially identified using the snowball process. One potential disadvantage of this recruitment strategy is that it may suffer from community bias (the potential for the first participants to impact the sample) [7]. The best defense against this is to begin with a group that is as diverse as possible [100]. Because we contacted our initial group of participants through multiple means (see Section 3.2.1.1), they show diversity by spanning organizational, product, and physical boundaries.

Our survey participants are drawn wholly from the populations of ABB, Google, IBM, and Microsoft. As a result, our findings may not be representative of the opinions and experiences of all practitioners. However, each of these companies is diverse and develops a myriad of products in various domains that run on a spectrum of platforms. In addition, we purposely targeted respondents from different areas and teams to increase heterogeneity and maximize generalizability.

We took care when creating our survey to address well-known design pitfalls [76]. In addition, we piloted the survey with a small initial set of practitioners and solicited suggestions on how to improve the survey. Based on their answers, we improved the survey before it was distributed, for example, by removing potential sources of ambiguity.

When conducting surveys through invitation, avoiding the self-selection principle is difficult [140]. Consequently, practitioners with responsibilities related to energy usage may see more benefit from contributing than others. Since we are primarily concerned with the perspectives of green software engineers, this is unlikely to represent a threat.

3.2.2 Findings

This section presents our findings from analyzing the data collected from both the interviews and the survey. Due to confidentiality requirements, we present only anonymized, aggregate information. At a high-level, we were interested in answering two main research questions:

- **RQ1:** In what domains is energy usage of concern to practitioners?
- **RQ2:** When energy usage is of concern, what are experienced practitioners' perspectives on green software engineering?

To answer the first question, we considered the responses of all survey respondents. To answer the second research question, we focused on the responses of *experienced practitioners*, the 176 survey respondents (40 %) who indicated that

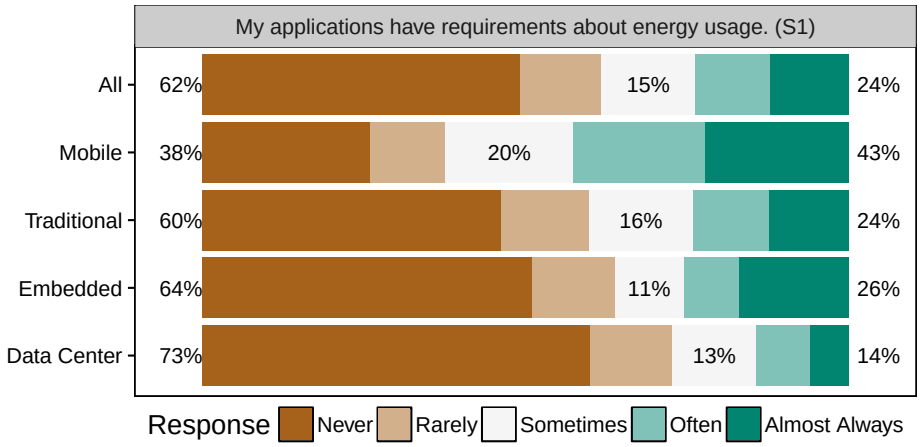

Figure 3.5: Responses to Statement S1 from all respondents and respondents who indicated that they are experienced in each domain.

their projects have energy usage requirements: Sometimes, Often, or Almost Always (see Section 3.2.2.1). Because we are interested in the perspectives of practitioners who have experience in green software engineering, including the responses of practitioners who are inexperienced would obscure the data of interest. The quotations presented in this section are taken from both interview transcripts and survey responses.

3.2.2.1 Where is Energy Usage a Concern?

To understand in what domains energy usage is of concern to practitioners, we first asked respondents how frequently they write code for applications that run on mobile devices, traditional PCs, data centers, and embedded platforms. Based on their responses, we consider a participant to be experienced in a domain if they write code for that domain Sometimes, Often, or Almost Always.

Next we asked respondents how frequently their applications have requirements about energy usage. Figure 3.5 shows a summary of the responses we received. The top of the figure shows the statement that was presented to participants followed by a label that we use to refer to the statement. The left-hand side of the figure shows different groups: *All* contains the responses of all 464 respondents; *Mobile* contains the responses of the 241 experienced mobile respondents; *Traditional* contains the responses

of the 328 experienced traditional respondents; *Embedded* contains the responses of the 47 experienced embedded systems respondents; and *Data Center* contains the responses of the 255 experienced data center respondents. The body shows stacked bar charts summarizing the proportion of respondents that chose, from left to right, Never, Rarely, Sometimes, Often, and Almost Always. The numeric percentages in the figure indicate the percentage of respondents that chose Never or Rarely (left), Sometimes (center), and Often or Almost Always (right). For example, for the *All* group, 62 % of respondents answered either Never or Rarely, 15 % answered Sometimes, and 24 % answered Often or Almost Always.

Based on our interviews, we initially theorized that practitioners with experience in mobile ("*battery life is very important, especially in mobile devices*"), data center ("*any watt that we can save is either a watt we don't have to pay for, or it's a watt that we can send to another server*"), and embedded ("*maximum power usage is limited so energy has a big influence on not only hardware but also software*") would more often have requirements or goals about energy usage than traditional practitioners ("*we always have access to power, so energy isn't the highest priority*"). However, the results from the survey only partially support this belief.

Experienced mobile practitioners frequently have requirements or goals about energy usage. As Figure 3.5 shows, our belief that mobile practitioners have goals or requirements about energy usage most often was confirmed by the survey: 63 % of experienced mobile practitioners responded that they have such requirements Sometimes, Often, or Almost Always.

Experienced traditional practitioners have requirements or goals about energy usage more often than expected. While we thought that Traditional projects would be by far the least likely to have energy requirements or goals, 40 % of experienced traditional practitioners indicated that they have energy requirements or goals Sometimes, Often, or Almost Always. One possible explanation for this finding is that there is overlap between experienced mobile developers and experienced traditional developers. More specifically, 58 % of the respondents in the Traditional group (189 out

of 328) have experience writing code for mobile devices (i.e., they indicated that they write code for mobile devices Sometimes, Often, or Almost Always). Because we asked about the frequency of energy requirements or goals for the respondent's applications in general, rather than for each domain, this level of mobile experience is likely shifting the range of responses to the more-frequent end of the spectrum. In future work, we plan to revisit this question by collecting more targeted data. Another possible explanation is that some traditional products also run on devices where battery-life is a concern (e.g., laptops).

Experienced embedded and experienced data center practitioners rarely have requirements or goals about energy usage. While we expected embedded and data center products to frequently have goals or requirements about energy usage, only 37 % of embedded respondents and 27 % of data center respondents indicated that they have such requirements more frequently than Rarely. Moreover, these responses are also likely shifted towards the more-frequent end of the spectrum: 52 % of embedded respondents (25 out of 48) and 37 % of data center respondents (95 out of 255) indicated that they also write code for mobile devices Sometimes, Often, or Almost Always.

For the data center group, the survey responses do not necessarily contradict our interview participants. Our interview participants were primarily responsible for the physical aspects of the data centers (e.g., computers, routers, power, cooling, etc.) while our survey primarily targeted the employees who are responsible for the services that run on the data centers. This difference suggests that while there are power and energy usage concerns at the lower levels, these concerns are not influencing the requirements and goals of the applications and services that run on the data centers. Both sides seem to agree on the cause of this disparity. One program manager summarized it as follows:

> *Our main concern is marketshare and that means user experience is a priority. We can be more efficient to try to cut costs, but since we don't charge by energy used this doesn't make us more attractive to users. So we tend to focus on other things like performance or reliability.*

For the embedded group, we found several reasons why practitioners do not

have requirements or goals about energy usage. First, many embedded products are not battery-powered (e.g., "*In our embedded systems, we always have access to power so energy is not a concern.*"). Second, many practitioners are concerned with the overall energy usage of their systems but rely on the hardware, not the software, to reduce energy usage. Finally, satisfying other metrics is more important than reducing energy usage (e.g., "*Ensuring the deterministic, real-time behaviour of our embedded device is more important than saving energy.*").

3.2.2.2 Perspectives on Requirements

Our survey statements concerning requirements focused on whether practitioners have experience with energy usage requirements (see Section 3.2.2.1), what typical energy usage requirements look like, and how often practitioners make tradeoffs between other features and energy usage.

Energy requirements are more often desires rather than specific targets. In addition to their Likert responses to Statement S1, we also asked experienced practitioners to provide an example of an energy requirement or goal. The majority of the provided examples are what we consider desires rather than detailed requirements. For example, one respondent said that they have "*no specific goals for energy usage, just 'don't be bad'.*" Another respondent indicated that "*considerations on background tasks as well as things that use the radios in phones are always in the back of my mind*" and a third stated that, "*the goal is to accomplish something without making the user annoyed about battery drain.*" Although desires are more common, detailed requirements do exist in some cases. An interesting example is: "*turn-by-turn guided navigation should not drain more battery than a car can charge.*" In addition, a few respondents indicated that they have had requirements similar to "*perform[ing] [user scenario] should not use more than* X mA" or "*under normal usage, a device with an* X W h *battery should last for* Y *hours.*"

Energy-usage requirements are often stated in terms other than energy usage. It is also interesting to note that many of the example goals and requirements

are expressed in terms of things other than battery life or energy usage. We believe that this is likely due to the lack of tool support for measuring energy usage (see Section 3.2.2.5). As a result, requirements are often written in terms of more easily captured metrics that practitioners believe correlate with energy usage. In some cases, these are traditional performance metrics. As one respondent stated: "*I don't usually think about battery life directly. Often I consider running time [...] and that 'seems' to suggest battery life.*" In other cases, they are countable events (e.g., "*We tried to optimize for when/how often we wake up the radio.*"). It is interesting to note that some practitioners are aware that such proxy measures may not be accurate:

> *Most people think power savings = CPU reduction. This is somewhat true in a broad sense, but is only a small part of the picture. The problem is that it's easy to measure CPU utilization (and hence reduction), but it's very hard to translate any of this to actual power savings. Many people have spent a lot of time that ultimately had no benefit.*

Unfortunately, this misconception is common and is an example of the levels of uncertainty that even experienced practitioners have (see Sections 3.2.2.4 and 3.2.2.4). Finally, there are requirements and goals that are defined in terms of previous or alternate versions, or as one respondent expressed it, "*'not worse than' kinds of requirements*" (e.g., "*New feature additions or architecture changes shouldn't regress battery life.*" and "*I had a requirement that energy usage in our primary scenario be comparable to the legacy solution.*").

Energy-usage requirements focus on "idle time." A common theme in our interviews and survey responses was the importance of reducing energy usage when a user is not interacting with their device. As one participant stated: "*We're trying to prioritize idle battery consumption down to zero. Being active is going to drain the battery. But the thing that's going to piss people off, is if I wasn't using it and my battery is dead so that's where we want to focus our efforts.*" In fact, one participant was so focused on idle time that they were surprised by the suggestion that non-idle time portions of an execution should also be optimized: "*I haven't thought about that, actually, when an app is in the foreground and we're trying to still save battery in some*

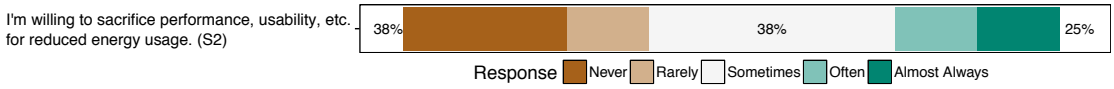

Figure 3.6: Requirements-related statement and responses from experienced practitioners.

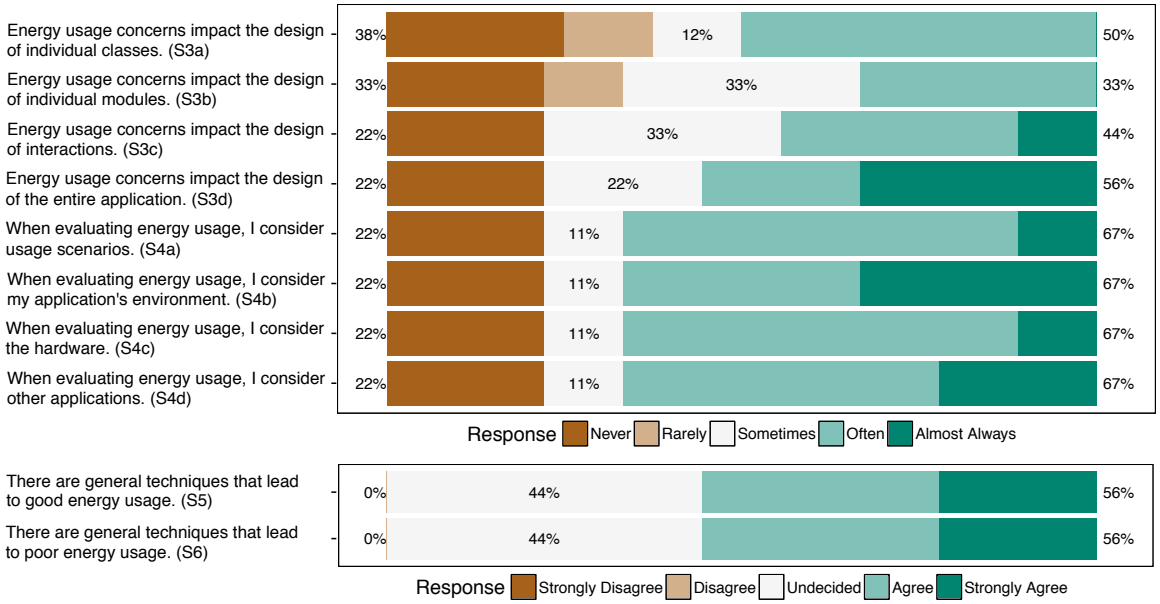

Figure 3.7: Design-related statements and responses from experienced practitioners.

way."

Practitioners are often willing to sacrifice other requirements for reduced energy usage. Figure 3.6 shows, in the same format at Figure 3.5, a summary of experienced practitioners' responses when asked how frequently they are willing to make tradeoffs between other requirements and energy usage. As the figure shows, respondents are overwhelmingly willing to make sacrifices to improve energy usage (80 % of respondents answered Sometimes, Often, or Almost Always). As several respondents stated: "*There is always a tradeoff between battery life vs performance/feature*" and "*the entire experience was a series of compromises between what designers wanted and [...] battery concerns.*" In fact, only 5 respondents answered that they Never make

such compromises.

3.2.2.3 Perspectives on Design

Our survey statements concerning design focused on how energy concerns impact different aspects of the design process, including the contexts that practitioners consider when assessing energy usage and the extent to which they believe there exist general patterns that lead to reduced energy usage and anti-patterns that lead to increased energy usage. Figure 3.7 shows the results that we collected for these statements using the same stacked bar format as earlier figures.

Concerns about energy usage impact how applications are designed. Based on our interviews, we believed that application design would be heavily influenced by energy concerns. As one participant stated: *"It's not a bug fix to get power efficiency. It's a design change."* The data for Statements S3a–S3d indicate that this sentiment is widely held; energy usage concerns frequently impact the design of individual classes, individual modules, interactions, and entire applications. Moreover, with the exception of individual classes, more than 50 % of respondents indicated that such impacts occur Sometimes, Often, or Almost Always. Interestingly, although individual classes are impacted less often, many practitioners believe that efficient algorithms, which are presumably implemented in a single class, are an effective way to reduce energy usage (see the discussion of patterns and anti-patterns below).

High-level designs are impacted by energy usage concerns more frequently than low-level designs. The responses for Statements S3a–S3d also show that 85 % and 81 % of respondents indicated that Sometimes, Often, or Almost Always, concerns about energy usage impact the design of interactions and entire applications, respectively. Conversely, only 47 % and 65 % of respondents indicated that concerns about energy usage impact the design of classes and modules Sometimes, Often, or Almost Always, respectively.

Practitioners consider usage scenarios most often when evaluating energy usage. The responses for Statements S4a–S4d show that 92 % of respondents consider

usage scenarios when evaluating energy usage Sometimes, Often, or Almost Always. Moreover, only 4 respondents indicated that they Never consider user scenarios when evaluating energy usage. The application's environment is the next most frequently considered context (79 % of respondents answered Sometimes, Often, or Almost Always), with hardware close behind (69 % of respondents answered Sometimes, Often, or Almost Always). These responses agree with our interviews. As one participant indicated, they have "*started looking at telemetry more*" in order to "*figure out more realistic goals*" and that "*there's a lot of other situations where we've tweaked little things here and there based on telemetry.*"

Practitioners consider other applications least often when evaluating energy usage. Unlike for Statements S4a–S4c, in response to Statement S4d, more respondents indicated that they considered other applications Never or Rarely (39 %) than Often or Almost Always (30 %). In total, 61 % of respondents indicated that, when evaluating energy usage, they consider other applications Sometimes, Often, or Almost Always. This suggests that practitioners may believe that interactions between applications are unlikely to impact energy usage or that such interactions are too numerous or difficult to consider.

Practitioners believe general patterns that lead to good or bad energy usage exist. The majority of respondents agree that there are general techniques that both lead to good energy usage (Statement S5, 55 % of respondents Agree or Strongly Agree while only 9 % Strongly Disagree or Disagree) and bad energy usage (Statement S6, 65 % of respondents Agree or Strongly Agree while only 3 % Strongly Disagree or Disagree). However, it is interesting to note that in both cases, there is a relatively large proportion of respondents who are Undecided (36 % and 32 %, respectively), which indicates that even experienced practitioners are unsure about whether such patterns exist.

To gain more information about the kinds of (anti-)patterns that practitioners believe exist, we asked respondents who responded with Agree or Strongly Agree to

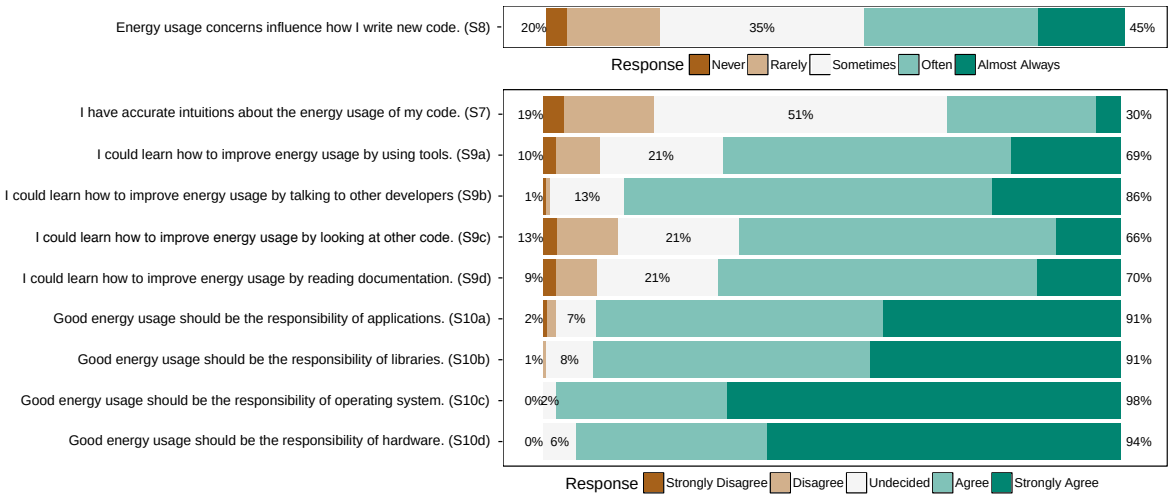

Figure 3.8: Construction-related statements and responses from experienced practitioners.

provide an example of such patterns. In general, each list of answers is the inverse of the other (e.g., for good energy usage do X; not doing X leads to poor energy usage). However, their responses show the complex tradeoffs that practitioners must make. For example, one participant stated that "*offloading computation to the cloud*", which requires using the radio to send and receive messages, is an effective method for reducing energy usage, while other participants noted that "*decreased radio use increases battery life.*" These tradeoffs mean that practitioners cannot blindly adhere to a set of rules, but must deeply understand the tradeoffs of the operations they are performing. Overall, the most frequently mentioned techniques for improving energy usage are: using an event-driven architecture instead of polling, coalescing timers, and using efficient algorithms. All of these techniques allow applications to achieve longer periods of inactivity, which matches the requirements-level focus on optimizing idle time energy usage and the design-level focus on optimizing interactions and entire applications.

3.2.2.4 Perspectives on Construction

Our survey statements concerning construction focused on learning whether energy concerns influence how new code is written, if practitioners believe that they have accurate intuitions about energy usage, how they would like to learn how to improve energy usage, and who they feel should be responsible for energy usage. Figure 3.8 shows the results that we collected for these statements using the same stacked bar format as earlier figures.

Energy concerns influence how practitioners write new code. The responses for Statement S8, show that 80 % of respondents consider energy concerns when they write new code Sometimes, Often or Almost Always. This result is the opposite of what we expected from our interviews where one participant said that "*Only when meeting performance goals becomes egregious in terms of power, then we negotiate a compromise that balances [...] performance and power consumption.*" Practitioners seem to take energy requirements and goals into consideration immediately, rather than waiting until energy issues are identified.

Practitioners believe that they do not have accurate intuitions about the energy usage of their code. The responses for Statement S7 indicate that, while 30 % of respondents believe that they have accurate intuitions about the energy efficiency of their code, the majority either disagree (19 %) or are undecided (51 %). This result matches our interview data. As one participant stated: "*I care about memory usage, CPU usage, like I understand those. [...] I don't have the same intuition about energy.*" This result also further supports our overall perception that, while practitioners have energy requirements, they lack the same level of expertise that they have with other types of requirements.

Practitioners believe that they could learn how to improve energy efficiency in many ways. The responses for Statements S9a–S9d show that practitioners are eager to learn how to improve the energy efficiency of their code in any way that they can. As one participant stated: "*I would love to have more education [...] for designing*

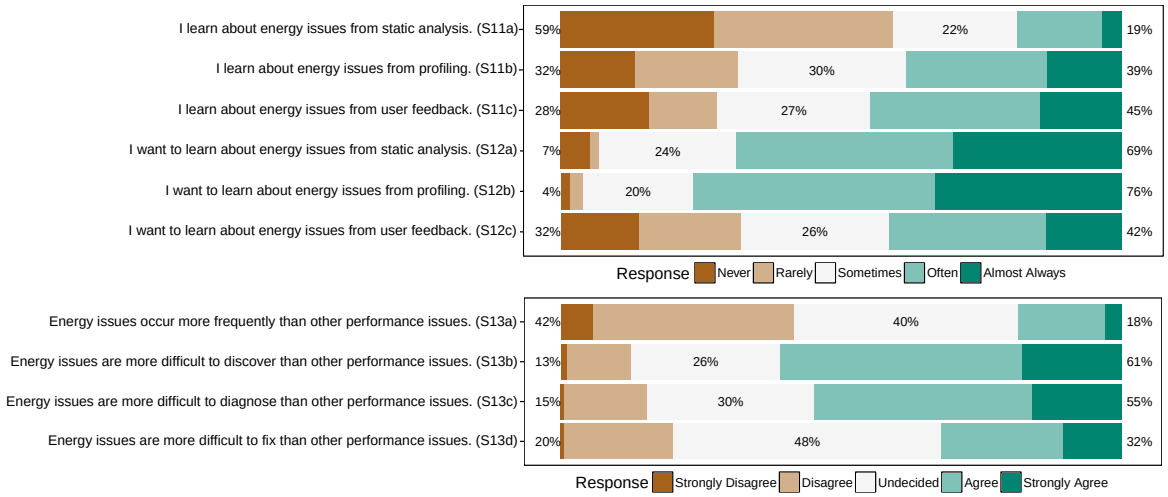

Figure 3.9: Finding and fixing issues-related statements and responses from experienced practitioners.

and investigating battery lifetime! Anything to help raise awareness and break through attitude barriers." Among the options that we specifically asked about, participants state they could learn from other developers (86 % of respondents answered Agree or Strongly Agree while only 1 % answered Strongly Disagree or Disagree) and feel that using tools, looking at other code, and reading documentation would be roughly equivalent in effectiveness (66 % to 70 % of respondents answered Agree or Strongly Agree for each option while 9 % to 13 % answered Strongly Disagree or Disagree).

Energy usage should be a shared responsibility. The responses for our final four statements, Statements S10a–S10d, show that respondents strongly believe that energy usage is a responsibility that is shared among applications, libraries, operating systems, and hardware. As one respondent stated: "*we are all in the same boat.*" Only 2 % of respondents Strongly Disagree or Disagree that applications have a responsibility for good battery life, and the percentage of respondents that Strongly Disagree or Disagree for the other elements is even lower. In fact, zero respondents Strongly Disagree or Disagree that operating systems and hardware should be responsible for good battery life.

3.2.2.5 Perspectives on Finding and Fixing Issues

Our survey statements concerning finding and fixing energy issues focused on learning how practitioners *currently* learn about energy usage issues (problems or defects related to energy use), how they would *prefer* to learn about those issues, how frequently energy issues to occur, and how difficult energy issues are to discover, diagnose, and fix. Figure 3.9 shows the results that we collected for these statements using the same stacked bar format as earlier figures.

Practitioners currently learn about energy issues primarily from profiling and user feedback. The responses for Statements S11a–S11b, show that practitioners currently learn about energy issues in their applications in a variety of ways. With 72 % of respondents answering Sometimes, Often, or Almost Always, the most common way they learn about such issues is by profiling performance metrics and counters (e.g., CPU usage). User feedback is a close second with 69 % of respondents answering Sometimes, Often, or Almost Always. While the frequency that practitioners learn about energy issues from profiling and user feedback was expected, the high number of respondents (41 %) that indicated that they learn about energy issues from static analysis Sometimes, Often, or Almost Always was surprising. In our interviews, few participants were aware of static analysis tools for detecting energy issues.

To learn more about the static analysis tools that these practitioners are using, we emailed the respondents who answered Sometimes, Often, or Almost Always to Statement S11a. We found that practitioners were using static analysis tools that do not identify energy issues directly, but rather look for code patterns (e.g., spawning lots of threads, polling frequently, bad data structures) that lead to bad CPU performance. Here, the practitioners are proceeding under the assumption that such metrics correlate with energy usage that we reported when discussing practitioners' perspectives on requirements (see Section 3.2.2.2).

Practitioners want to learn about energy issues most frequently from profiling and static analysis. While static analysis is currently the least used technique

for learning about energy usage issues, 93 % of respondents indicated that they would Sometimes, Often, or Almost Always like it to be effective (Statement S12a). However, although many respondents are enthusiastic—"*Having static analysis to point out deficiencies of efficiency would be awesome*"—some are skeptical about its feasibility—"*Good luck getting static analysis to work on this.*" The ability to detect energy issues via profiling is also highly desired with 96 % of respondents indicating that they would Sometimes, Often, or Almost Always like it to be effective (Statement S12b). Finally, despite the fact that user feedback is currently one of the most commonly used approaches, practitioners are least enthusiastic about it. Although they would rather learn about issues than have them go undetected (68 % of respondents answered Sometimes, Often, or Almost Always), it appears they would prefer to learn about such issues earlier in the development process, before users are impacted.

Practitioners are unsure, but suspect that energy issues do not occur more frequently than other performance issues. The responses for Statement S13a indicate that, while 42 % of respondents Strongly Disagree or Disagree with the statement, nearly as many (40 %) are Undecided about whether energy issues occur more frequently than other types of performance issues. It is possible that this perception is true, however it may also be a reflection of the fact that there are few tools capable of detecting such issues and only the most egregious problems are reported by users.

Practitioners believe that energy issues are more difficult to discover and diagnose than other performance issues. The responses to Statements S13b and S13c indicate that respondents believe that energy issues are more difficult to discover than performance issues (61 % of respondents answered Agree or Strongly Agree) and more difficult to diagnose than performance issues (55 % of respondents answered Agree or Strongly Agree). When asked to explain why they have these beliefs, respondents provided a wide range of answers. Many of them feel that energy issues are more difficult to discover because "*current test suites are not equipped for them,*" they are "*not sure what tools exist to discover such issues,*" and "*performance issues are very*

obvious—the application is slow, frozen, etc.—but battery drain is a slower change and is not as immediately noticeable." Similarly, many respondents felt that energy issues are difficult to diagnose because there are "*too many variables that affect power*", "*the [observable] problem is far removed from the source*", and energy issues "*are most often emergent behaviors arising from complex interactions between many subsystems rather than found in one subsystem.*"

Practitioners are undecided about whether energy issues are more difficult to fix than performance issues. The responses for Statement S13d indicate that the majority of respondents (48 %) are Undecided about whether energy issues are more difficult to fix than performance issues. Again this might be true, or it might be because practitioners have not fixed enough energy issues to form an overall impression of their difficulty. The respondents who agreed that energy issues are more difficult to fix primarily felt this way because if "*it was not considered from the start, improving battery life or energy usage could require large changes*" or could "*require some high level re-design.*" These reasons match our observations in Section 3.2.2.4 that energy concerns most often impact high-level designs. Respondents also felt that in many cases fixes are difficult because the problem is outside of their control (e.g., "*Dependencies on libraries [...] that are inherently inefficient can make battery life issues hard to improve.*" and "*Problems don't always reside in the app code. The hardware often doesn't support polling, idle, or other modern commands to minimize energy usage.*").

3.2.2.6 Perspectives on Maintenance

Our survey statements concerning maintenance focused on learning how practitioners take energy concerns into consideration when making changes, and documenting and reviewing code. Figure 3.10 shows the results that we collected for these statements using the same stacked bar format as earlier figures.

Energy concerns are largely ignored during maintenance. The responses for Statements S14–S17 indicate that participants are the least concerned with energy when performing maintenance activities. For each statement, the largest number of

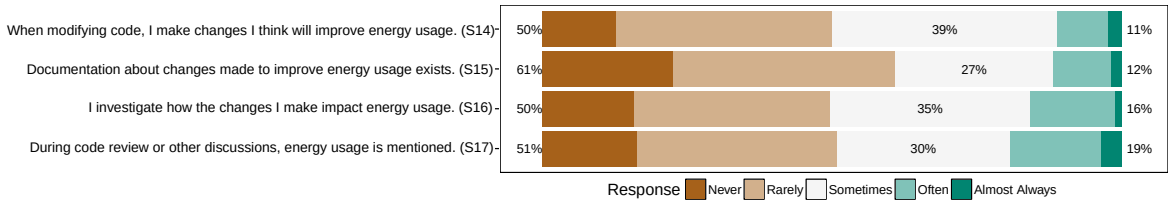

Figure 3.10: Maintenance-related statements and responses from experienced practitioners.

respondents answered Never or Rarely. The lack of tool support that we identified in Section 3.2.2.5 likely explains why respondents do not investigate the impacts of the changes that they make. It is less clear however, why respondents are not creating documentation or discussing energy with other developers, when they feel that these would be effective ways of learning how to improve the energy efficiency of their code (see Section 3.2.2.4).

3.2.3 Implications

In the remainder of this section, we (1) contextualize the state-of-the-art in green software engineering research with respect to our study's findings, and (2) suggest directions for researchers aiming to develop strategies and tools to help practitioners improve the energy usage of their applications by addressing practitioners' lack of information and support infrastructure.

3.2.3.1 Implications for Requirements

The findings that "energy-usage requirements are often stated in terms other than energy usage" and "energy usage requirements are more often desires rather than specific targets" suggests that energy requirements are difficult to specify directly. Existing work on eliciting quality or "just in time" requirements (e.g., [45, 47]) may serve as a starting point. A potential hurdle to extending such work is the lack of an easily understood energy metric. Providing easy to use energy measurement tools may help in this situation, but the concept of a joule is likely to remain too abstract. An approach

proposed by Zhang et al. supports "not worse than" requirements by advocating the use of benchmark scenarios to compare energy usage between alternatives. A strength of this approach is its focus on scenarios, which practitioners are already considering (see Section 3.2.2.4). However, it requires portable benchmarks and alternative implementations, which may be unavailable. **Requirements elicitation strategies would be more useful if they helped practitioners easily understand and express how much energy usage is reasonable for a given task.**

The finding that "experienced practitioners are often willing to sacrifice other requirements for reduced energy usage" motivates the development of techniques and tools for exploring potential tradeoffs between energy usage and other non-functional requirements. Again, existing work in the area of trade-off analysis may provide a starting point [154], but no one has investigated whether such approaches are suitable for energy usage requirements. **Techniques for quantifying how changes in energy usage impact other quality requirements such as performance would help practitioners make intelligent trade-off decisions.**

3.2.3.2 Implications for Design

The finding that "practitioners consider usage scenarios most often when evaluating energy usage" is promising. It suggests that the large body of scenario-based research (e.g., [104, 128]) is applicable when designing energy-efficient software. The focus on scenarios also suggests that some scenarios are more sensitive to energy efficiency than others. **Tools and techniques will be more valuable to practitioners if they are scenario-aware.**

The findings that "practitioners believe general patterns that lead to good or bad energy usage exist" and "high-level designs are impacted by energy usage concerns more frequently than low-level designs" motivates empirical studies of the impacts of such patterns. While there has been a significant amount of work in understanding how changes made by developers impact energy usage (e.g., [83, 89, 96, 115, 124, 131, 132]), the considered changes have been at a lower level than the (anti-)patterns suggested

by the practitioners. The few studies that have considered higher-level decisions (e.g., design patterns [130], web servers [96]) are preliminary and limited in scope. **Studies that evaluate whether practitioners' beliefs are correct and provide context to help decision making can have an impact on design.**

3.2.3.3 Implications for Construction

The finding that "energy concerns influence how practitioners write new code" suggests that new programming languages or language features could help developers during the development of energy-efficient applications. Existing work in the area of energy-aware programming (e.g., [107, 159]) matches practitioners' focus on idle time by allowing computation to be degraded or delayed to save energy. However, to use such features effectively, practitioners must understand the energy impacts of their code, which given their lack of intuition (see Section 3.2.2.4), may be infeasible. **Practitioners' focus on idle time motivates additional investigation into new programming paradigms for delaying computation and bundling work as well as automated transformations for improving energy usage.**

The finding that "practitioners believe that they do not have accurate intuitions about energy usage" motivates the creation of energy profiling tools that can help them understand the energy usage of their applications. Researchers have proposed several such tools (e.g., [58, 62, 134]), but most of these approaches are coarse-grained, which may limit their usefulness. One exception is work by Li et al. which attempts to calculate source line level energy information [84]. In general, fine-grained energy profiling is difficult for many reasons including high clock rates and tail energy. Tail energy is particularly challenging since it means that software energy usage can depend on other applications, the factor least considered by practitioners (see Section 3.2.2.4). **Fine-grained tools supporting whole system analysis would help practitioners understand their code and the energy impacts of interactions among applications.**

The finding that "practitioners believe that they could learn how to improve energy efficiency in many ways" suggests that they lack the necessary knowledge, expertise, and intuition about how to construct energy-efficient software. The desire to learn is evident in their responses, which indicate that they would use all forms of help that we asked about (other developers, tools, profiling, example code, documentation, etc.). **Education mechanisms in any and all forms would likely be received well by practitioners**.

3.2.3.4 Implications for Finding and Fixing Issues

The finding that "practitioners believe that energy issues are more difficult to discover than other performance issues" motivates techniques for detecting energy issues. Existing static analysis-based work (e.g., [115]) is comparable to the tools currently used by practitioners (see Section 3.2.2.5) because they look for patterns that may lead to energy issues (e.g., forgetting to close a resource). Existing testing or dynamic analysis-based work (e.g., [11, 85, 89, 92, 93, 150]) attempts to locate energy issues directly, but is limited by imprecise oracles. **Practitioners would like oracles that can (1) detect energy issues as they occur, rather than waiting for battery drain to become evident, and (2) determine whether the amount of energy being consumed is reasonable given the work being performed.**

The finding that "practitioners believe that energy issues are more difficult to diagnose than other performance issues" motivates the need for new techniques for debugging energy issues. To the best of our knowledge, no one has yet investigated such approaches. **Debugging techniques should take into consideration the large distances between when and where faulty behaviors are discovered and the root causes of such issues.**

3.2.3.5 Implications for Maintenance

The finding that "energy concerns are largely ignored during maintenance" demonstrates the importance of focusing on energy use in earlier phases of the development life cycle. Presumably, once an application enters maintenance, it is either too difficult or not important enough to change energy usage. The lack of documentation regarding energy usage and the low number of respondents who investigate how their changes impact energy usage may point to a need for improved practices and tooling, but further study is needed to understand why energy appears to be ignored during maintenance and what tooling or practices can help. To the best of our knowledge, no one has investigated what roles energy usage concerns play during maintenance. **Additional surveys and interviews may help uncover why energy concerns are ignored during maintenance.**

3.2.4 Related Work

The work presented by Pang et al. is the closest to our study. In [114], Pang et al. have studied green software engineering practitioners perspectives on Construction [114]. Compared to their work, our study is broader in several ways: (1) We considered four additional phases of the development cycle that led to unique observations for the Requirements, Design, Finding and Fixing Issues, and Maintenance phases, (2) The participants of our study include data center and embedded practitioners in addition to mobile and desktop practitioners, and (3) We interviewed more practitioners and collected a larger number of survey responses. As a result, our data reflects the perspectives of a larger group and provides more evidence that the results are representative.

Beyond studying green software engineering practitioners directly, there is work that shares our desire to understand practitioners' perspectives on various aspects of the software development process and suggest areas for future research. One set of work examines the adoption of tools for specific development tasks. For example, Johnson et al. analyzed the reasons why software engineers do not use static analysis

tools for automatic code inspections [73]; and Cherubini et al. investigated how developers use drawings to represent code [22]. Other researchers have examined particular development activities. For example, de Souza and Redmiles proposed an analytical framework about developers' strategies to handle the effect of software dependencies and changes [33]; Dagenais et al. studied and characterized project landscapes for newcomer developers [29]; and Murphy-Hill et al. analyzed the differences between video game development and other software development [103].

3.2.5 Summary

In the second part of this chapter, we presented the methodology, findings, and implications from our study of practitioners' perspectives on green software engineering. We interviewed 18 professional practitioners from Microsoft, and surveyed 464 developers and testers from ABB, Google, IBM, and Microsoft, to collect data that would help us identify practitioners' practices and challenges presented during the software development process when trying to make their applications more energy efficient.

The results of this study indicate that:

(1) Experienced mobile and traditional practitioners frequently have requirements or goals about energy usage.

(2) Energy requirements are more often desires rather than specific targets. These requirements are often stated in terms other than energy usage, and the majority are focus on idle time.

(3) Practitioners are often willing to sacrifice other requirements for reduced energy usage.

(4) Concerns about energy usage impact how applications are designed, where high-level designs are impacted by energy usage concerns more frequently than low-level designs.

(5) When evaluating energy usage, practitioners consider usage scenarios more often than other applications.

(6) Practitioners believe there exist general patterns that lead to good or bad energy usage.

(7) Energy concerns influence how practitioners write new code. However, practitioners believe that they do not have accurate intuitions about the energy usage of their code.

(8) Practitioners believe that they could learn how to improve energy efficiency in many ways. They consider that energy usage should be a shared responsibility among applications, libraries, operating systems, and hardware.

(9) Practitioners currently learn about energy issues primarily from profiling and user feedback. However, they want to learn about energy issues most frequently from profiling and static analysis.

(10) Practitioners are unsure, but suspect that energy issues do not occur more frequently than other performance issues.

(11) Practitioners believe that energy issues are more difficult to discover and diagnose than other performance issues, but they are undecided about whether energy issues are more difficult to fix than performance issues.

(12) Energy concerns are largely ignored during Software maintenance.

Chapter 4

THE SOFTWARE ENGINEER'S ENERGY DECISION SUPPORT FRAMEWORK (SEEDS)

Analogous to other software improvement goals, e.g., performance, power consumption should be amenable to improvement beyond that achievable by low-level systems and hardware. Unfortunately, few software developers design and implement applications with consideration for their energy usage. Results from our study of practitioners' perspectives on green software engineering revealed that this is due to two primary reasons: (1) developers do not understand how the software engineering decisions they make affect the energy consumption of their applications, and (2) they lack the tool support to help them make decisions or change their code to improve its energy usage.

This chapter describes SEEDS, a novel framework to help software engineers develop energy-efficient applications without having to address the low-level, tedious work of applying changes and monitoring the resulting impacts to the energy usage of their application. SEEDS provides automated analysis, decision-making, and implementation of decisions towards optimizing a given targeted software engineering decision with regard to energy usage of the entire application. SEEDS also takes into account the execution context (i.e., platform and expected inputs) where the application will be deployed.

4.1 Overview of SEEDS

We designed SEEDS to support three primary goals:

(1) automate the entire process of optimizing the application with respect to potential code changes to save developers from performing tedious, error-prone tasks,

(2) abstract away the systems and hardware platform interactions from developer concern, and

(3) be general enough to support different types of decisions commonly made by software engineers, including optimization goals, filtering mechanisms, search strategies, energy profiling approaches, and hardware platforms.

4.1.1 Example Usage Scenario

Amy is a developer who would like to evaluate how different Collections implementations used by her application can help her to reduce her application's energy usage, thus Amy decides to use an instantiation of SEEDS called the $\text{SEEDS}_{\text{api}}$ to perform this task. Amy provides to $\text{SEEDS}_{\text{api}}$ the following inputs: (1) the application source code in a JAR file, (2) the application's test suite, (3) a set of alternative Collections libraries (e.g., fastutil, and javolution) packed as JAR files, and (4) the list {Set, List} to specify the optimization parameters, i.e., which Collections types she wants to focus the energy optimization on. After providing the inputs, she runs $\text{SEEDS}_{\text{api}}$ to automatically test all alternative Collection implementations and find all the points in her application where Set and List implementations have been instantiated and could be replaced by alternative implementations found in the fastutil and javolution Collections libraries. $\text{SEEDS}_{\text{api}}$ then checks if the transformations of alternative Set and `List` implementations do not violate the syntax of the application, and also checks if these transformations comply with the application's test suite (i.e., pass all the tests). $\text{SEEDS}_{\text{api}}$ helps Amy to assess the energy usage of instantiating multiple and different Collections implementations in various locations of her application by creating different versions of her application that contain different transformations involving the instantiation of Set and List implementations from the Collections libraries specified by Amy. As an output, $\text{SEEDS}_{\text{api}}$ gives Amy the name and location of the Set and List transformation(s) suggested for her application to make it more energy efficient, and the energy savings obtained from the corresponding transformations. With the information provided as an output by $\text{SEEDS}_{\text{api}}$, Amy avoids manually exploring locations in her

```
168     public synchronized Set getAvailableIDs() {
169         // Return a copy of the keys rather than an unmodifiable collection.
170         // This prevents ConcurrentModificationExceptions from being thrown by
171         // some JVMs if zones are opened while this set is iterated over.
172         return new TreeSet(iZoneInfoMap.keySet());
173     }
```

Figure 4.1: Place where a Set Implementation is instantiated and can be switched with another Set implementations.

application where Collections implementations are used and need not manually test all different possible combinations to obtain the set of Collections implementations that helps her application to be more energy efficient; all this work in done by $\text{SEEDS}_{\text{api}}$ automatically.

Specifically, consider a simplified example from the Jodatime project [1]. Suppose Amy intends to replace the use of `java.util.TreeSet` with other Set implementations such as `it.unimi.dsi.fastutil.objects.ReferenceOpenHashBigSet`, `javolution.util.FastSet`, and `java.util.concurrent.CopyOnwriteArraySet`. Amy would like to switch Set implementations in all locations where a Set instance is used in her project, and then assess the energy usage impacts of using different Set implementations. In the Jodatime project for instance, Amy could switch the Set implementation in line 172 shown in Figure 4.1. Without SEEDS, to evaluate how the transformations of Set implementations impact the energy efficiency of her application, Amy has to manually perform the following tasks: (1) Find additional Set implementations from alternative collections libraries (e.g., fastutil, javolution and goldman sachs collections), (2) Pinpoint points in the application, such as the one shown in Figure 4.1, where the alternative Set implementations could be used, (3) Transform the application so that an alternative Set implementation is used instead of the one instantiated in the original version of the application, e.g., instantiate `java.util.concurrent.CopyOnwriteArraySet` in line 172 of Figure 4.1, instead of `java.util.TreeSet`, (4) Check if the applied transformation(s) are syntactically and semantically equivalent to the original

[1] http://www.joda.org/joda-time/

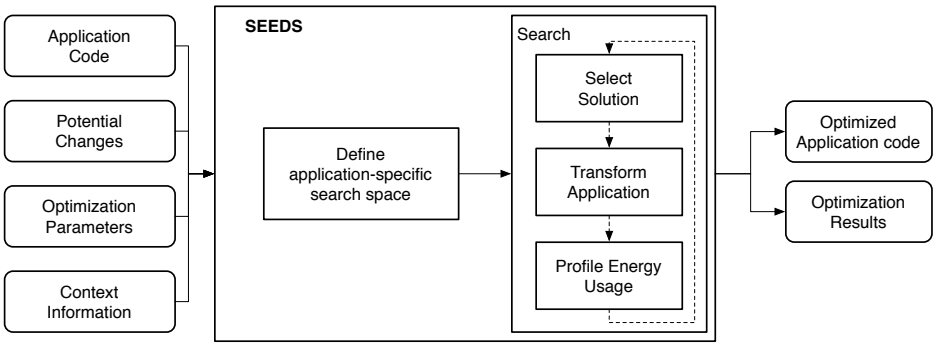

Figure 4.2: Overview of SEEDS.

application, (5) Run the application with the applied transformation(s) to measure its energy consumption, with some kind of energy measurement device or prediction, (6) Compare the energy usage of the original and transformed version of the application and select the application version with the lowest energy usage, and (7) Repeat steps (1)-(6) with different transformation of Set implementation(s) until all combinations have been tested or an energy efficient transformation have been found. Steps (1) through (7) could be done manually by Amy for a small set of locations and alternative implementations. However, when the number of locations and alternative implementations grows, this process of locating, transforming and profiling the energy usage of an application is error-prone and arduous. Thus, instead of doing it manually, Amy can use $\text{SEEDS}_{\text{api}}$ to automatically evaluate which transformations in terms of Set implementations could help her application be more energy efficient. By using $\text{SEEDS}_{\text{api}}$, Amy can automatically test all Collections implementations from different Collection libraries in 15 different points of the Jodatime application where Collection instances are used. With the results of $\text{SEEDS}_{\text{api}}$, Amy learns that the transformation to make in line 172, shown in Figure 4.1, is to switch from a `TreeSet` implementation to a `ReferenceOpenHashBigSet` implementation to save 9% of energy usage when running the Jodatime project.

4.1.2 SEEDS Components

Figure 4.2 provides a high-level overview of SEEDS. In the remainder of this section, we provide a detailed discussion of each of the framework's main components.

4.1.2.1 SEEDS Inputs

As shown in Figure 4.2, SEEDS requires four inputs: the *application code*, a set of *potential changes*, the developer's chosen *optimization parameters*, and additional *context information*.

The **application code** is the code of the application that the software engineer wants to optimize with regard to energy usage. The **set of potential changes** includes all of the changes that the developer is deciding whether or not to make. For example, the set of potential changes could include decisions such as which library implementation to use, whether to perform refactoring, whether to replace an algorithm with a different algorithm, or whether to cache the result of a computation. Note that the transformations specified in the set of potential changes are abstract rather than concrete (e.g., inline a method vs. inline method `foo` in method `bar` at line 5). This allows sets of changes to be specified and frees developers from the task of re-specifying them for each new application that they want to improve. The process for transforming abstract potential changes into concrete changes for the given application is described in Section 4.1.2.3.

The **optimization parameters** are constraints on where SEEDS should consider making changes. For example, a developer could restrict the application of a refactoring to only certain regions of the application or only allow switching algorithms if the algorithm's inputs are larger than a given threshold. Optimization parameters are defined in terms of the features of both the set of potential changes and the locations where those changes can be applied in the application. Thus, optimization parameters can include guidance about how changes should be applied. For example, the software engineer could provide a ranking of alternative library implementations for a given API based on their intuition about the performance of one implementation over

another. Or, they may be considering various refactorings based on recommendations from a tool or documentation that indicates that applying that refactoring improves code readability and maintainability.

Finally, the **context information** indicates the platform where the application will be executed, the expected inputs or workload that will be used to drive the application, and other relevant data about the application needed by the optimization strategy. The strategy used for energy profiling (see Section 4.1.2.4) dictates the specific information that needs to be provided. For example, if energy profiling is to be done using a hardware-based platform, then the platform itself and a set of suitable, concrete inputs are needed. However, if energy profiling is to be done using a dynamic analysis-based estimation approach, then execution traces and a model of the platform are required. Finally, if a static estimation approach is used, a developer may not have to provide any context information at all. The ability to provide context information can be especially useful if a developer does not have easy access to a target system. Essentially, they can "cross-optimize" (in the same spirit as cross-compilation) their application to a wide range of target platforms.

Given these inputs, the key tasks of SEEDS are then to:

(1) Define the application-specific search space, that is, the space of concrete changes that SEEDS will consider, and

(2) Search through the application-specific search space to find an optimized version of the application that reduces energy usage as much as possible.

Each of these tasks is described in more detail in the following subsections using as reference the class diagram of SEEDS shown in Figure 4.3. This figure shows the SEEDS framework as a white-box framework, i.e., a framework that relies on inheritance and dynamic binding for its extensibility.

Algorithm 1 SEEDS algorithm

```
procedure SEEDS(appJARFile, testSuiteName, changes[], optPar[])
    origApp ← new AppSolution(appJARFile)
    optConst ← Constraint.setOptParameters(optPar)
    setChanges ← Transformation.getSetOfChanges(changes)
    codeIR ← AppGeneralizer.generalize(appJARFile, setChanges)
    AppGeneralizer.identifyAppSites(codeIR, setChanges)
    AppLocation setSites[] ← AppGeneralizer.filterSites(optConst)
    rewriter← new Rewriter(codeIR)
    sSpace ← newSearchSpace()

    for site in setSites do
        for change in setChanges do
            if site.IsTransformationApplicable(change) then
                if AppGeneralizer.checkSite(change, testSuiteName) then
                    sol ← new AppSolution(site, change)
                    sSpace.addSolution(sol)

    eProfiler ← new EnergyProfiler()
    solSelector ← new SolutionSelector(sSpace)
    searchAlg ← new SearchAlgorithm(origApp, solSelector, rewriter, eProfiler)
    best ← searchAlg.findBestSolution()
     return best
```

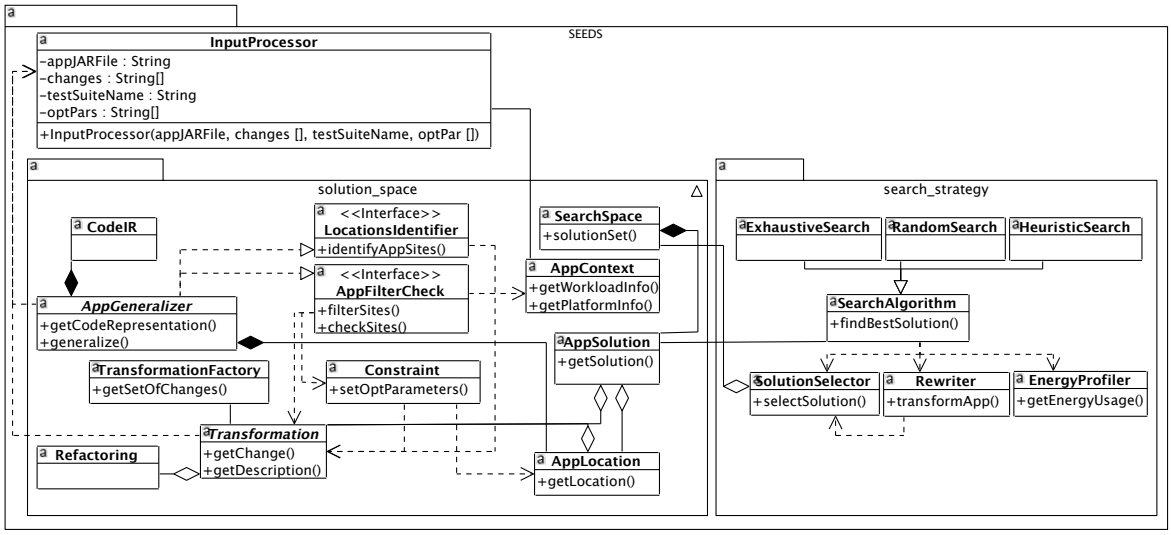

Figure 4.3: SEEDS Framework Class Diagram.

4.1.2.2 SEEDS Outputs

The output of SEEDS includes the optimized application code, the optimization results that contain the transformations made to the original application, and the energy savings obtained from such transformations. The optimized application code generally is an optimized version of the given application where the energy usage is reduced as much as possible with respect to the given set of potential changes, optimization parameters, and context information. Note that the optimized version is not guaranteed to be optimal with respect to all possible versions (i.e., there may be another version of the application that uses less energy when given different parameters). When given a specific combination of inputs, the application may not be improved in terms of energy usage. Then, SEEDS returns the original application version as output. In our evaluation, this situation occurred twice. Although this is not our desired outcome, knowing that an application can not be improved by SEEDS is useful information. It indicates that the developer is free to make any of the considered changes without needing to consider how they would impact energy usage, and also informs

the developer that other different strategies may be tested in order to possibly reduce the energy usage of the application. Algorithm 1 presents the general algorithm for SEEDS in terms of the specification provided in Figure 4.3.

4.1.2.3 Application-Specific Search Space in SEEDS

In Figure 4.3, the package `solution_space` contains all classes and interfaces of SEEDS that are related to the creation of the application-specific search space. The transformations made by SEEDS are selected from the set of potential changes, which are abstract rather than concrete i.e., they are specified in terms of a super type that can be interchangeable with a different set of sub types, and they do not specify a specific program point to be applied. A potential change of the application is represented in SEEDS by the `Transformation` class, thus creating sub classes of `Transformation` allows definition of sub types representing potential changes. As shown in Figure 4.2, SEEDS' first task is to determine the application-specific search space by concretizing these abstract changes with respect to the given application and optimization parameters. In essence, the application-specific search space is the set of all possible versions of the given application that could be created by SEEDS. The `SearchSpace` class is in charge of creating and maintaining the set of possible versions of the given application, where each version is represented by an instance of the `AppSolution` class.

To create the application-specific search space, SEEDS considers each potential change and scans the application's code to identify the locations where the change could be applied. For example, if a potential change is to inline a method, SEEDS will identify all of the locations in the application where a method is invoked. This initial list of concrete changes is then filtered based on both explicit and implicit constraints. Explicit constraints are generally based on the type system of the programming language used to implement the application. For example, implementations of an API can only be swapped if both implementations expose the same interface. Implicit constraints are most commonly provided by the SEEDS user as optimization parameters.

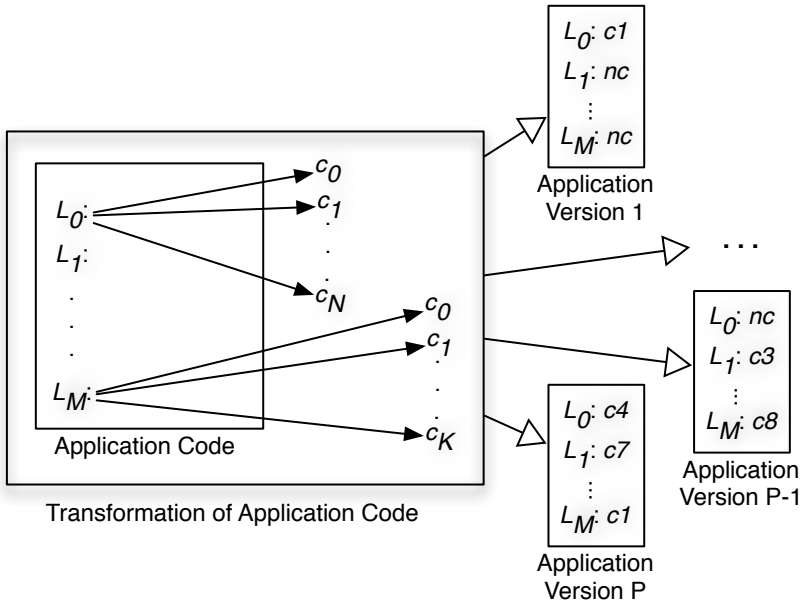

Figure 4.4: Application versions belonging to the application-specific search space.

A location is represented in SEEDS by the `AppLocation` class, and the interfaces `LocationIdentifier` and `AppFilterCheck` provide the base behavior to identify locations in the application code and filter location sites, respectively.

Figure 4.4 shows an example of the application versions belonging to the application-specific search space in SEEDS. In this figure, the application code contains M change locations where concrete changes can be made. At each change location L_i, a change c_j can be applied, or not (nc). Each change location has a set of potential changes that can be applied. For instance, at location L_0, a total of N different changes can be applied, and at location L_M, a total of K individual changes can be applied. Three types of application versions belonging to the search space are shown in Figure 4.4. Application version 1, with change c_1 applied at change location L_0 while leaving the other change locations without changes, represents application versions in the search space that are generated by applying only one change at any of the identified change locations of the application code. Application version P-1 represents another type of application version where multiple change locations are transformed by changes, in this

case change, locations L_1 and L_M were transformed by changes c_3 and c_8, respectively. Finally, application version P represents application versions having a change at every change location in the application code.

4.1.2.4 SEEDS Search

The package `search_Strategy`, in the right side of Figure 4.3, includes the classes and interfaces corresponding to the selection, transformation and profiling of solutions from the search space in SEEDS. Depending on the given set of potential changes and number of locations in the application where those changes can be correctly applied, the application-specific search space could be very large. Manually exploring such a space would be a tedious, error-prone task for a software engineer, and furthermore such a space may actually be too large to search exhaustively and would require some kind of sampling as in search-based software engineering.

SEEDS' search task is responsible for navigating the application-specific search space to find versions of the application that consume less energy than the original version, and ultimately choose the optimized version of the application that results in the greatest amount of energy savings. As shown in Figure 4.2, at a high level, the task of searching the application-specific search space is divided into three steps: (1) *select* a solution from the application-specific search space (i.e., a concretized change or set of changes), (2) *transform* the original application by applying the chosen solution, and (3) *profile* the energy usage of the transformed version.

The search process begins by selecting a solution from the search space. In practice, essentially any selection strategy could be employed. The minimum requirement of the selection algorithm is that it guarantees that the selected application solution is syntactically and semantically correct. The `SolutionSelector` class is in charge of implementing the selection methodology. For example, the selection strategy could be to select a solution in a random manner, based on a heuristic, or using a genetic algorithm. The search task is represented in Figure 4.3 by the `SearchAlgorithm` class and its subclasses `ExhaustiveSearch`, `RandomSearch` and `HeuristicSearch`. One of

the main benefits of defining SEEDS in this way is that software engineers can tailor the selection component to their specific applications and sets of potential changes. Essentially, the selection component can be specified by creating a subclass of the `SolutionSelector` class in SEEDS.

The second step of the search task is to transform the application by applying the chosen solution. The class `Rewriter` is in charge of this transformation process. Often such transformations will be done using support provided by an integrated development environment (IDE) or other stand-alone tools. Although SEEDS attempts to filter out invalid concrete changes when creating the application-specific search space, it is possible that unknown, implicit constraints are broken by applying the changes. For example, the application may assume, but not document, that the iteration order of a collection must be fixed or that elements are returned in sorted order. If concrete inputs or an explicit test suite is provided as part of the context information, SEEDS can perform regression testing to address this possibility. Regression testing ensures that, with respect to the provided inputs, a modified version of an application is semantically identical to its original version. Transformed application versions that fail regression testing are simply discarded and a new solution is chosen.

The third step of the search task is to profile the transformed version of the application to calculate its energy usage. This could be achieved through existing techniques such as the hardware-based, simulation-based, or estimation-based approaches presented in Section 2.1. The `EnergyProfiler` class, shown in Figure 4.3, provides an abstraction of the technique to profile the energy usage of the application's versions in SEEDS.

After calculating the energy usage of the transformed version, the search process begins again. The selection step incorporates the new information about how the selected solution impacts energy usage and chooses a new solution. The transform step applies the new solution and, if possible, checks whether it produces a valid application. And the profile step calculates the energy consumption of the new, transformed application version. The search process iterates in this fashion until a stopping point

is reached. Similar to how any selection strategy can be used, any stopping criterion can be used. The search could stop when the energy usage of the application has been reduced by a certain percentage or is less than a given threshold. It could stop after a specified number of iterations or when energy usage does not improve for a given number of iterations. The stopping criterion could also halt the search after a set amount of time or when the application-specific search space has been completely explored.

We explore using two types of search strategies that can be implemented in an instantiation of SEEDS: an exhaustive search strategy and a search strategy based on metaheuristic optimization.

4.1.2.4.1 Exhaustive Approaches

Exhaustive search strategies try to find an improved solution to a problem through the exploration of all possible solutions in the search space. Here, we present the general exhaustive search strategy and a limited exhaustive search focused on exploring not all, but some, sections of the search space in SEEDS.

Exhaustive Search

An exhaustive strategy in SEEDS allows an exploration of all possible versions of an application based on the specified potential changes in all possible locations of changes in an application. As shown in Figure 4.4, three kinds of application versions can be explored by the exhaustive search strategy when applying potential changes in change locations of an application. However, considering and applying the combination of all possible changes in all possible change locations of an application creates a very large space of application versions that is difficult to explore in terms of the time required to generate and evaluate every application version. As a result, an instantiation of SEEDS can explore one of the simplest search strategies possible: a limited exhaustive exploration of all possible versions of an application created by applying a single concrete change at each identified change location of the application where the potential set of changes can be applied.

Limited Exhaustive Search

The limited exhaustive search strategy starts by identifying the most energy efficient implementation choice at each location in the program where a change can be made. In general, an instantiation of SEEDS can identify the most energy efficient change at each location by applying each concrete change to the program, running the resulting version multiple times, and comparing the means of the energy usages to find the change that results in the least amount of energy consumption. After identifying the most efficient implementation at each application's change location, the corresponding SEEDS instantiation creates one additional version where the most efficient change at each change location is applied. Recalling the application versions belonging to the search space shown in Figure 4.4, the limited search strategy explores application versions having a single change at one change location (e.g., like application version 1), and application versions having a single change at every change location in the application (e.g., like application version P). After all application versions have been generated, the selection strategy compares the energy impacts of all of the versions executed during the search process and then it selects as the output of the tool the application version that results in the largest decrease in energy usage. If none of the application versions is a statistical improvement over the original, unmodified application, the original application version is returned instead. In this way, a SEEDS instantiation is guaranteed to never produce an application version that performs worse than the original application.

4.1.2.4.2 Metaheuristic-based Search

Metaheuristic optimization comprises a set of algorithms and techniques that employ some degree of randomness to find optimal, or near-optimal, solutions to hard problems [94]. More specifically, metaheuristics are used to solve problems for which an optimal solution is unknown beforehand, there is no clear way to find an optimal solution, and an exhaustive or brute-force search is very expensive to use because the size of the search space is too large. Because all of the aforementioned situations apply

to our problem of finding an energy efficient version of an application, we decided to investigate evolutionary algorithms, the most popular kind of metaheuristic strategies, as a search strategy in SEEDS.

Evolutionary algorithms are based on the metaphor of the biological evolution theory [42, 94]. In the context of evolutionary algorithms, the individuals subjected to evolution are the solutions for a given problem, and these solutions belong to the search space of the optimization problem. In terms of the number of objectives, there are two types of evolutionary algorithms that can be considered: *single objective* and *multiobjective* optimization algorithms. In single objective optimization algorithms, the goal is to find a single solution for which a single objective is improved. Multi-objective optimization algorithms consider the case of the simultaneous presence of several objectives that often contradict each other. More specifically, multiobjective optimization algorithms try to find one or multiple "optimal" solutions that can achieve the best optimization with the possibly conflicting objectives. Then, the goal is to find one or more solutions that belong to the set of optimal values of the problem, which is called the *pareto-optimal set*. The optimal values of the problem, which constitute the *pareto front*, are the collection of the objective vectors that cannot be dominated. In a minimization problem, an objective vector v is considered to be better, or to *dominate* another objective vector w in the pareto sense, if all the components of v are lower or equal to the components of w, with at least one strictly lower component. A *pareto-optimal set* is the collection of solutions from the search space with objective vectors belonging to the pareto front.

We decided to investigate metaheuristic-based search strategies to explore the search space of application versions (i.e., solutions) because metaheuristics (1) enable SEEDS to explore the search space in an intelligent and guided fashion, (2) they allow more sections of the search space to be explored, which is represented by the diverse set of application versions that consist of a combination of changes each one applied to different change locations in an application, and (3) there exist different types of metaheuristics approaches (i.e., single and multi-objective) that can be investigated to

find an improved version of an application.

4.2 Instantiating SEEDS for Software Engineers' Decisions

The design of SEEDS enables the creation of different instantiations to support the analysis of the energy usage of several common decisions that software engineers make. Here, we present the design of three instantiations of SEEDS. The first instantiation, called SEEDS_{dp}, supports the transformation of an application in terms of standard design patterns. SEEDS_{dp} can help developers to discover which design patterns to include or remove from his/her application to improve the application's energy efficiency. The second instantiation called $\text{SEEDS}_{\text{mobile}}$, seeks to assist mobile developers in examining and finding the combinations of different (already studied) transformations that are known to boost the energy efficiency of mobile applications. Finally, we present a third instantiation called $\text{SEEDS}_{\text{api}}$ that is designed to support developers when making decisions about which library implementations to use to make their applications more energy efficient.

4.2.1 Support for the Selection of Energy Efficient Design Patterns

The goal of the SEEDS_{dp} instantiation is to help developers to improve the energy usage of their applications by introducing or removing Design Patterns (DP) [48] in their applications. DP are solutions to commonly recurring problems in software design. Recent studies have shown that replacing some design patterns in an application can help to reduce the application's energy usage [24, 91, 108, 130]. For instance, in [130], we presented an empirical study where introducing the Decorator design pattern in an application increases the energy usage by about 700%, and by introducing the Flyweight design pattern, the energy reduces by 58%. Most recently, in [108], the authors proposed modifications to two different design patterns to reduce applications' energy usage.

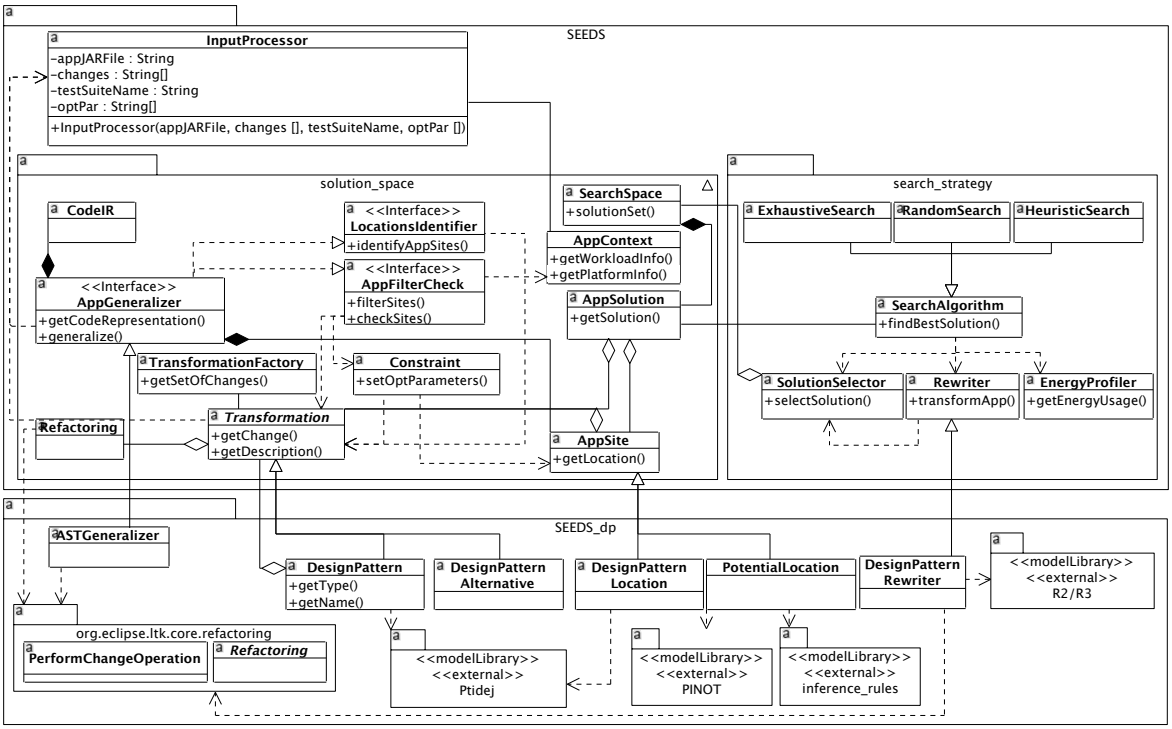

Figure 4.5: Class Diagrams for the SEEDS_{dp} instantiation (bottom) and the SEEDS framework (top).

Figure 4.5 shows the class diagram for the SEEDS_{dp} instantiation of SEEDS. This SEEDS_{dp} instantiation uses the general framework but specializes some components of SEEDS to comply with the new transformations analyzed in this instantiation. Here, we describe how some components of SEEDS are specialized in SEEDS_{dp}; other components are used as they were presented in Section 4.1.2. The potential changes are represented by instances of the `DesignPattern` class. To localize where DP instances can be replaced or introduced, two types of locations for DP can be identified by SEEDS_{dp}. The first kind of change location is the one related to an existing DP in the application code, which is represented by the `DesignPatternLocation` class. In general, DPs can be identified in an application by using some application metrics [26], and can be represented by a set of refactorings [75]. Existing tools such as Ptidej[2] or

[2] http://www.ptidej.net/tools/designpatterns/

ePAD [32] can be integrated to support the DP identification process. The second kind of change location that SEEDS_{dp} can identify is an application location where a DP can be introduced; this type of location is represented by the `PotentialLocation` class. Potential locations can be recognized by using strategies such as inference rules [72] or automated scripts [75]. By pinpointing current instances of DPs in the application, SEEDS_{dp} can replace such instances by either other DPs or DP alternatives i.e., equivalent solutions that can be used to substitute a DP [4]. Identifying potential locations for a DP with SEEDS_{dp} opens the possibility of using new DPs into the application with the aim of being more energy efficient.

Given a representation for DPs and their locations in an application, SEEDS_{dp} can generalize the application code so that existing DPs or potential locations can be detected. The `ASTGeneralizer` class holds an Abstract Syntax Tree (AST) representation of the application code. As DP transformations can be composed of a set of refactorings, the AST representation of the application code facilitates the application of refactorings associated with DP changes in the code. Finally, the `DesignPatternRewriter` class is extended from the `Rewriter` class in SEEDS to support the transformation of the application in terms of the corresponding DP changes selected during the search process.

4.2.2 Support for the Selection of Energy Efficient Refactorings for Mobile Applications

Several code transformations for mobile applications have been examined and proposed to reduce an application's energy usage [86, 87, 90]. For example, changing color configurations [90], grouping HTTP requests [86], and using adequate asynchronous tasks [87], are some of the changes that can be made to a mobile application to reduce its energy usage. General code transformations such as those associated with common refactorings (e.g., extract method, introduce parameter object), or with performance directed changes (e.g., avoid using getters/setters, or avoid using array

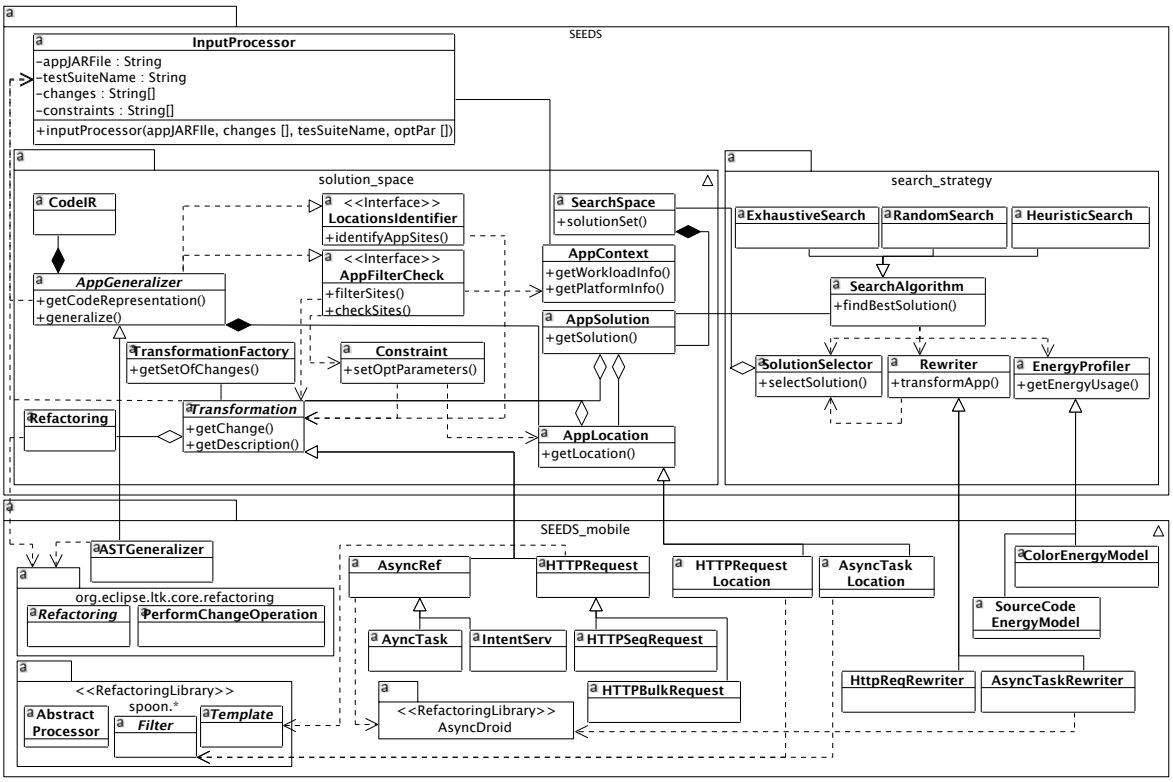

Figure 4.6: Class Diagrams for $\text{SEEDS}_{\text{mobile}}$ (bottom) and SEEDS (up).

length in loop body) can also be considered [83, 131]. The $\text{SEEDS}_{\text{mobile}}$ instantiation of SEEDS aims to support transformations to mobile applications that boost the application's energy efficiency. Because diverse interactions can exist between the portions of code being transformed, and there can be many possible combinations of transformations that can be made in an application's code, it is important to analyze the energy impact of the application of different combinations of code transformations. Hence, $\text{SEEDS}_{\text{mobile}}$ can help the developer to automatically explore and pinpoint which combination of such transformations is the most appropriate to improve the energy usage of an application.

In Figure 4.6, the class diagram of $\text{SEEDS}_{\text{mobile}}$ is depicted. Similar to $\text{SEEDS}_{\text{api}}$ and SEEDS_{dp}, some specializations of classes in SEEDS are made to conform to the

changes and localization of changes in $\text{SEEDS}_{\text{mobile}}$. For changes in the application code, the classes `AsyncRef` and `HttpRequest` represent two different types of changes related to the selection of appropriate asynchronous constructs to use in the application and the set of HTTP requests to group, respectively. The `HTTPRequestLocation` and `AsyncTaskLocation` classes are in charge of locating the points where HTTP requests and asynchronous constructs are located, respectively. For generalizing the application code, the `ASTGeneralizer` class can be used when transformations can be considered in terms of code refactorings. In this case, the `ASTGeneralizer` class provides an AST representation of the source code that facilitates the application of the corresponding refactorings in the application code. Finally, the `HTTPReqRewritter` and `AsyncTaskRewriter` classes provide a specialization of the `Rewriter` class in SEEDS to modify the application code according to the changes selected during the search process. For $\text{SEEDS}_{\text{mobile}}$, tools like AsyncDroid [87], for asynchronous tasks, and Spoon [118] for other refactorings, can be leveraged to support $\text{SEEDS}_{\text{mobile}}$ tasks.

4.2.3 Support for the Selection of Energy Efficient Library Implementations

One of the most common decisions that developers make on a daily basis is the selection of Collections implementations, which provide general data structures and algorithms to store and manage different kinds of data objects used in their applications. $\text{SEEDS}_{\text{api}}$ is an instantiation of SEEDS that supports software engineers as they make decisions about which library implementations they should use to optimize the overall energy usage of their applications. More specifically, $\text{SEEDS}_{\text{api}}$ optimizes Java applications by identifying implementations of the Java Collections API that are more energy efficient, if any, than the implementations currently used by the application. We chose to actually implement $\text{SEEDS}_{\text{api}}$ to evaluate SEEDS. The next section describes our $\text{SEEDS}_{\text{api}}$ instantiation and implementation.

4.3 Case Study: $\text{SEEDS}_{\text{api}}$ Instantiation and Implementation

To evaluate SEEDS, we chose to target the choice of the Collections API implementations in $\text{SEEDS}_{\text{api}}$ for several reasons. First, choosing a collection implementation is a common decision that is faced by developers. Second, in many cases, developers are choosing API implementations based on familiarity or execution time concerns. This means that applications are unlikely to have optimized their choice of collection implementation to energy usage. Finally, the impact of Collections API choice has not been investigated by researchers. As such, investigating their impacts supports our goal of using the framework to explore the energy optimization space and enable researchers to answer questions that they could not previously ask.

Because we were the first to look at the choice of API implementations, we conducted a study to determine whether the choice of API implementations did in fact impact the energy usage of an application, before actually implementing $\text{SEEDS}_{\text{api}}$. After showing the results of our study, we present how we instantiated $\text{SEEDS}_{\text{api}}$.

4.3.1 Observational Study of Collections API Impact on Energy

Before implementing $\text{SEEDS}_{\text{api}}$, we performed a preliminary study to determine if changing implementations of the Collections API could indeed impact the energy usage of an application. To answer this question, we created 13 versions of a publicly available micro-benchmark.[3] At a high-level, this benchmark creates an instance of a class that implements the `Collection` interface and then performs a large number of operations on the instance (e.g., adding single elements, adding another collection of elements, removing some elements, removing all elements, etc.). We chose to use this benchmark for two reasons. First, it has previously been used to evaluate the runtime performance of implementations of the Collections API. Second, it is a micro-benchmark; The majority of its execution is spent in the code of the collections implementations. This allows us to focus directly on our area of interest (i.e., the collections implementations).

[3] http://java.dzone.com/articles/java-collection-performance

Table 4.1: Potential Improvement or Degradation in Energy Usage from Switching Collection Implementations.

	Potential Gain from Switching		Potential Loss from Switching	
Current Choice	# Better	Max Improvement (%)	# Worse	Max Degradation (%)
ArrayList	2	95	0	—
ConcurrentLinkedQueue	4	96	0	—
LinkedHashSet	0	—	7	2,598
HashSet	0	—	7	2,617
LinkedList	5	96	0	—
TreeSet	0	—	5	1,974
PriorityQueue	2	96	0	—
ConcurrentLinkedDeque	6	96	0	—
CopyOnWriteArrayList	0	—	2	79
ConcurrentSkipListSet	0	—	4	1,495
LinkedBlockingDeque	6	96	0	—
LinkedTransferQueue	5	96	0	—
CopyOnWriteArraySet	0	—	5	1,602

Each of the 13 versions of the benchmark that we created uses a different concrete implementation of the `Collection` interface. The first column of Table 4.1, *Current Choice*, shows the 13 concrete implementations of the `Collection` interface that we considered. We then executed each version of the benchmark 10 times and profiled its energy usage using the LEAP system (See Section 4.3.2.5 for a detailed explanation of how we profile energy usage.) We then conducted pair-wise statistical analysis of the versions' energy usage using the Kruskal-Wallis test. Essentially, we determined, given a current implementation choice, whether switching to another implementation decreases, increases, or has no effect on energy usage.

In Table 4.1, the second and fourth columns, *# Better* and *# Worse* show, given the current implementation in the first column, the number of times switching to another implementation improves energy usage ($\alpha = 0.05$) and the number of times switching to another concrete implementation worsens energy usage ($\alpha = 0.05$), respectively. For example, if the currently selected implementation is `ArrayList`, there

are two implementations that will decrease the benchmark's energy usage and no implementations that will increase the benchmark's energy usage. As the table shows, for 7 of the 13 cases, energy usage can be statistically improved by switching implementations, and for 6 of the 13 cases, energy usage can be statistically worsened. These results show that indeed switching implementations of the Collections API can in fact impact the energy usage of an application.

To gain some additional insight into the effects of switching implementations, we investigated the magnitude of the increases and decreases. For the cases where there is a statistically better or worse alternative implementation, we calculated the percentage difference in the mean energy usage of the 10 runs for the current version and the mean energy usage of the 10 runs of the best alternative and the worst alternative. Note that for this benchmark, `HashSet` is the most energy efficient implementation and `LinkedBlockingDeque` is the most inefficient implementation.

In Table 4.1, the third column, *Max Improvement*, and the fifth column, *Max Degradation* show the percentage change from switching from the current version to the best version and from the current version to the worst version, respectively. A dash (—) indicates a case where there was not a statistically better or worse choice than the current implementation. For example, switching from `ArrayList` to `HashSet` results in nearly a 100 % improvement in energy usage while switching from `LinkedHashSet` to `LinkedBlockingDeque` increases energy usage by over 2,500 %. Not only does switching implementations of the Collections API statistically significantly impact energy usage, but the magnitude of the impact can be quite large. These empirical results quantify the potential impact of a framework such as SEEDS.

4.3.2 $\text{SEEDS}_{\text{api}}$ Components

Motivated by the results from our preliminary study on the impact of switching implementations of the Collections API, we created the $\text{SEEDS}_{\text{api}}$ instantiation of SEEDS. $\text{SEEDS}_{\text{api}}$ inherits and uses the functionality provided by SEEDS to create an instantiation of the framework that finds the application's energy-efficient

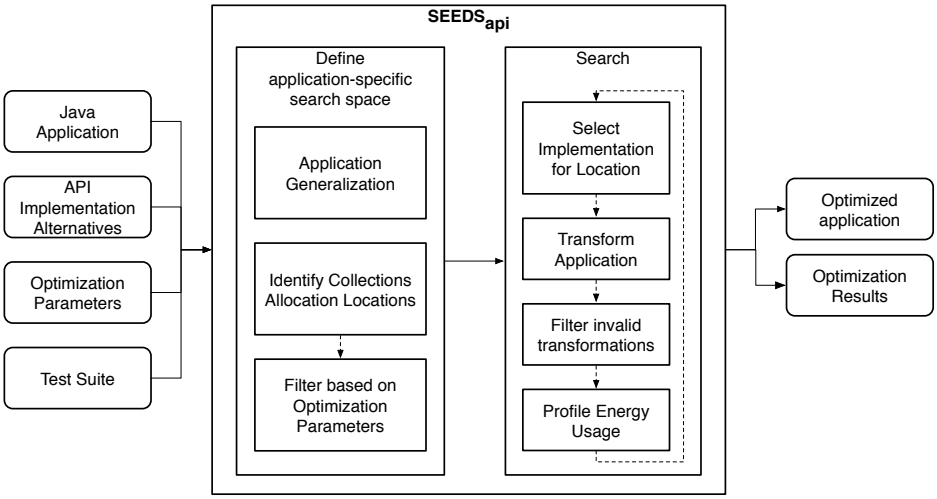

Figure 4.7: Overview of SEEDS$_{\text{api}}$.

transformations composed of API implementations. The remainder of this subsection describes how each of the components of SEEDS was instantiated in SEEDS$_{\text{api}}$.

A high-level overview of SEEDS$_{\text{api}}$ is shown in Figure 4.7, and the remainder of this subsection describes how each of the components of SEEDS was instantiated in SEEDS$_{\text{api}}$. Components that are not specifically mentioned were implemented as described in Chapter 4.

In Figure 4.8, the class diagrams and the associations for both SEEDS$_{\text{api}}$ and SEEDS are shown.

4.3.2.1 SEEDS$_{\text{api}}$ Inputs

The `InputProcessor` class in Figure 4.8 captures the inputs required by SEEDS$_{\text{api}}$: application code, potential changes, context information and optimization parameters.

Application code. SEEDS$_{\text{api}}$ is designed to optimize Java applications. Therefore, it accepts as input Java applications that use the Collections API; this input is represented by the `appJARFile` property of `InputProcessor`.

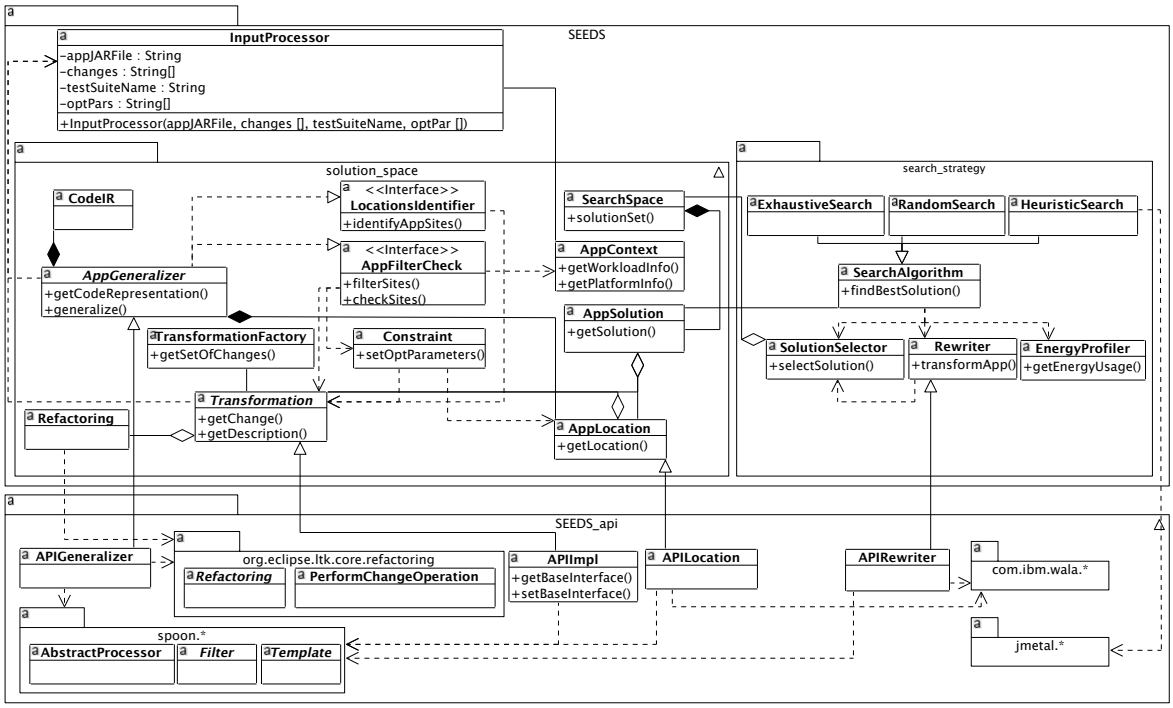

Figure 4.8: Class Diagrams for $\text{SEEDS}_{\text{api}}$ (bottom) and SEEDS (top).

Potential Changes. The set of potential changes indicates which implementations of the Collections API can be substituted for one another. For example, a potential change would be to substitute `HashSet` for a `TreeSet` or vice versa or `LinkedList` for `ArrayList`. A potential change in $\text{SEEDS}_{\text{api}}$ is represented by an instance of the class `APIImpl`; $\text{SEEDS}_{\text{api}}$ obtains the set of libraries from which to extract the set of potential changes from the `changes` property of the `InputProcessor` class. Note that $\text{SEEDS}_{\text{api}}$ can consider changes between any implementations that implement the same Collections API. Currently, the tool includes all implementations from the Java Collections Framework (JCF) as well as all implementations of `Collection` from

Javolution,[4] fastutil,[5] Apache Commons Collections,[6] Goldman-Sachs Collections,[7] and Google's Guava libraries.[8] If developers would want to consider additional potential changes (e.g., implementations from another library), they can simply provide the library's jar file to our tool via the `InputProcessor` class.

Context information. To use $\text{SEEDS}_{\text{api}}$, developers must provide a test suite as part of the context information. The property `tesSuiteName` of `InputProcessor` stores the name of the application's test suite. The test suite is used to perform regression testing to ensure that all considered transformations are valid and to execute the transformed applications during profiling.

Optimization Parameters. Developers can also provide a list of optimization parameters that help them specify constraints during the search process in $\text{SEEDS}_{\text{api}}$. Optimization parameters can include, for instance, API implementations or application locations on which to focus the search. The property `optPars` of `InputProcessor` allows developers to set the optimization parameters in $\text{SEEDS}_{\text{api}}$ by providing a list of String values obtained from the `Constraint` class in SEEDS.

4.3.2.2 $\text{SEEDS}_{\text{api}}$ Outputs

The output of $\text{SEEDS}_{\text{api}}$ includes the the Java energy efficient application version, the optimization results that contain the Collections implementations changes applied to the original application, and the energy savings obtained from such transformations.

---

[4] http://javolution.org

[5] http://fastutil.di.unimi.it

[6] http://commons.apache.org/proper/commons-collections

[7] https://github.com/goldmansachs/gs-collections

[8] https://code.google.com/p/guava-libraries/

4.3.2.3 SEEDS$_{\text{api}}$ Application-specific Search Space

SEEDS$_{\text{api}}$ uses the elements in the `solution_space` package of SEEDS (See Figure 4.8) to construct the search space of solutions containing different API implementation transformations.

We observed that, in many cases, developers do not "program to the interface", rather they specify a concrete type for their variables (e.g., `ArrayList l` vs. `List l`). Unfortunately for SEEDS$_{\text{api}}$, this practice can unnecessarily constrain the size of the application-specific search space and hinder the optimization process. To address the problem, SEEDS$_{\text{api}}$ *generalizes* the application's code by changing the type of each variable, for which original type is a subclass of `Collection`, to the most general supertype. For example, the type of a variable that was declared as a `LinkedList` could be generalized to: (1) `Collection` if only methods declared in the Collections interface are used, (2) `List` if methods declared in `List`, but not in `Collection`, are used (e.g., `get`), or (3) `LinkedList` if methods declared by `LinkedList`, but not `List`, are used (e.g., `addLast`). In practice, SEEDS$_{\text{api}}$ uses the Eclipse IDE's refactoring tools to automatically apply the Generalize Type refactoring to every variable where the type is a subclass of `Collection`. This functionality is represented by the class `APIGeneralizer` in SEEDS$_{\text{api}}$.

After generalization, SEEDS$_{\text{api}}$ analyzes the application to identify the locations where instances of the Collections API are created. Pinpointing locations of Collections instances is implemented in the `LocationsIdentifier` interface shown in Figure 4.8. Although this may seem like a trivial task, in Java, object allocation is actually a two-step process. First, a new instance of the desired type is created using the `new` bytecode. Then, at some point later, one of the type's constructors is invoked on the new object using the `invokespecial` bytecode. Identifying both bytecodes is necessary to be able to transform the application because the type of the object created by the `new` must be the same as the declaring type of the constructor that is invoked by the `invokespecial`. Unfortunately, there is no guarantee that the `new` and `invokespecial` are easy to match. In fact, depending on the structure of the code,

there can be an arbitrary number of intervening instructions. In order to identify pairs of `new` and `invokespecial` bytecodes that constitute an object allocation, $\text{SEEDS}_{\text{api}}$ uses the T.J. Watson Libraries for Analysis (WALA) [146] to implement a def-use analysis that tracks backwards from the target object of the `invokespecial` to the `new` where it was created. Locations suitable for transformations are represented in $\text{SEEDS}_{\text{api}}$ by the `APILocation` class, a subclass of the `AppLocation` class in SEEDS.

After identifying the locations in an application where collection objects are allocated, $\text{SEEDS}_{\text{api}}$, determines how many potential changes could be applied at each location. For example, if an instance of the Set interface is being created, $\text{SEEDS}_{\text{api}}$ identifies all potential changes that switch implementation to an implementation of the Set API. The combination of all allocation locations and possible changes, away from the generic type of the object being created, constitutes $\text{SEEDS}_{\text{api}}$'s application-specific search space, represented by an instance of the `SearchSpace` class. This instance maintains a set of `AppSolution` instances containing information about the changes and location of changes to be made in the application.

4.3.2.4 $\text{SEEDS}_{\text{api}}$ Search

Because there has been no prior investigation into the impacts of switching implementations of the Collections API on energy usage, we have no intuition or information on the shape of the application-specific search space or how to search through it effectively. For example, we have no idea if there are likely to be many local minima, if the effects of multiple changes are likely to be additive or independent, or even if the search space is differentiable. As a result, we implemented the exhaustive search strategy presented in Section 4.1.2.4, and a metaheuristic search strategy using evolutionary algorithms.

4.3.2.4.1 Limited Exhaustive Search

In the limited exhaustive search strategy, class `ExhaustiveSearch`, the procedure identifies the most energy efficient implementation choice at each location in the

program where an object that implements the Collections API is created. Locations instantiating a Collections object are called change locations. $\text{SEEDS}_{\text{api}}$ identifies the most efficient implementation choice at each change location by applying each concrete change (i.e., a Collection implementation) to the program, running the resulting version multiple times, comparing the means of the energy usages to find the change that results in the least amount of energy consumption. Application versions generated in this process have one change at one change location, similar to application version 1 in Figure 4.4. After identifying the most efficient implementation at each object allocation location, $\text{SEEDS}_{\text{api}}$ creates one additional application version where the most efficient change at each change location is applied, as application version P shown in Figure 4.4.

Although this strategy is simple, it results in optimized applications that are more energy efficient than the original applications. In our experiments, $\text{SEEDS}_{\text{api}}$ improved the energy usage of 7 of our 9 subject applications by between $\approx$2 % and $\approx$17 %. Moreover, as we explained before, this search strategy is meant to be a starting point for future research rather than the best way of creating optimized applications.

4.3.2.4.2 Metaheuristic-based Search

We selected evolutionary algorithms, more specifically Genetic Algorithms (GA), as the metaheuristic strategy to find energy efficient versions of an application. A Genetic Algorithm (GA) [42, 94] is an evolutionary algorithm that searches for an optimal, or near-optimal, solution by sampling the search space and by maintaining a set of solutions (i.e., *population*) to treat and analyze simultaneously. The population evolves through a series of iterations called *generations*. Solutions, a.k.a. individuals, belonging to a population are chosen from the search space by a given *selection strategy*, and then some solutions from the population are combined and mutated to evolve and create offsprings, until a new generation of solutions is created. Both combination and mutation operations are essential *genetic operators* that mix and introduce new material within a population, respectively. The selection of solutions for the next generation

is done from the parent population and the offspring population in accordance with a survival strategy that normally supports fit individuals i.e., solutions that perform very well with regard to a selected quality indicator.

Generational Genetic Algorithm (gGA)

The generational Genetic Algorithm (gGA) is an evolutionary algorithm for single objective optimization problems. This algorithm finds optimal solutions to a given problem by exploring the search space based on ideas similar to those in biological evolution theory [42]. We use a gGA as a search strategy in $\text{SEEDS}_{\text{api}}$ to explore the search space looking for solutions i.e., application versions that improve the energy usage of applications. The gGA first constructs an initial population and then iterates in a cycle composed of three major steps: (1) compute an objective and a fitness value for all the individuals in the population, (2) use the objective and fitness information of the individuals to breed a new offspring population by using the crossover and mutation genetic operators, and (3) join the parents and offspring to form a new next-generation population. The breeding process continues until the desired size of the offspring population has been reached. The phenomenon where there are no individuals in the population that are getting fitter than previous individuals is known as the bloat effect [94]. To avoid the bloat effect, we select the best two individuals in terms of their objective value (i.e., *elite* parents) from the previous population to be carried forward. Thus, the elite parents are selected to survive for the next offspring generation of individuals. The general cycle continues until a given number of generations has been completed. The total number of generations in which populations are being created is restricted by the selected stopping criteria: when a given total number of evaluations of the objective function have occurred or a total number of generations is reached, when a given objective value has been achieved by the best current solution, or when a given execution time has been reached.

Non-dominated Sorting Genetic Algorithm (NSGA-II)

To consider not only the energy usage of solutions but also their execution time when searching for an improved version of an application, we use a multiobjective evolutionary algorithm. In a multiobjective evolutionary algorithm, the GA maintains a diverse set of solutions that helps it to find multiple pareto-optimal solutions as described in Section 4.1.2.4.

The Non-dominated Sorting Genetic Algorithm II (NSGA-II) is an evolutionary algorithm for multiobjective optimization problems that is able to find better solutions than other multiobjective evolutionary approaches [34]. The NSGA-II differs mainly from another single objective GA in that the NSGA-II: (1) assigns the fitness values to population members based on a fast nondominated sorting procedure, and (2) tries to preserve diversity among solutions of the same nondominated front using elitism within solutions. By using a fast nondominated sorting procedure, the NSGA-II reduces the computational cost of the algorithm when trying to find an improved solution that considers multiple objectives at the same time for the problem under analysis. Furthermore, maintaining a good spread of solutions during the execution of the algorithm avoids that the algorithm gets stuck in a local optimal.

To implement the metaheuristic-based search strategies in $\text{SEEDS}_{\text{api}}$ using the gGA and NSGA-II algorithms, we leveraged the features provided by the JMetal [9] framework. JMetal is a widely adopted metaheuristic framework in Java [43] that provides different metaheuristic algorithms and can be easily integrated in $\text{SEEDS}_{\text{api}}$.

4.3.2.5 $\text{SEEDS}_{\text{api}}$ Transform and Profile

Transform application. Transforming the application's code, based on an `AppSolution` instance, is implemented in the `APIRewriter` class in $\text{SEEDS}_{\text{api}}$. To apply the selected changes, `APIRewriter` uses ObjectWeb's ASM bytecode rewriting library[10] to change the types of `new` and `invokespecial` bytecodes that correspond

[9] http://jmetal.sourceforge.net

[10] http://asm.ow2.org

to the locations of the selected changes. We chose to directly modify the application's bytecode, rather than its source code, so that $\text{SEEDS}_{\text{api}}$ does not have to recompile the application each time a change is applied. After each time the application is transformed, the test suite provided as input is used to ensure that the transformation has not broken the functionality of the application.

Profile energy usage. To profile the amount of energy consumed when executing an application, we considered two energy profiling techniques: A hardware instrumentation technique using a LEAP node [141], and an energy estimation technique using the RAPL interface [27].

Figure 4.9 shows our LEAP node, an x86 platform based on an Intel Atom motherboard (D945GCLF2). It is currently configured with 1 GB of DDR2 RAM, a 320 GB 7200 RPM SATA disk drive (WD3200 BEKT), and runs XUbuntu 12.04. Each component in the LEAP system (e.g., CPU, disk drives, memory, etc.) is connected to an analog-to-digital DAQ card (National Instruments USB-6215) that samples the amount of power consumed by the component at a rate of 10 kHz (≈10,000 samples per second). The LEAP also provides running applications with the ability to trigger a synchronization signal. This allows for synchronizing the recorded power samples with the portions of the execution that are of interest.

Note that while the original LEAP specification calls for using the same computer to both run an application of interest and collect power samples, we have modified the design to use dedicated hardware for each of these roles. Using separate machines prevents the introduction of any unwanted measurement overheads. The only remaining source of unwanted overhead is the collection of synchronization information. It is possible to account for this cost by profiling the energy cost of recording synchronization information and subtracting it from the reported energy numbers. However, because we are concerned with energy consumption relative to a base line (i.e., the original application) and the energy cost of recording synchronization information is essentially constant, we have not taken this step.

For the energy estimation technique, we used the RAPL interface available in

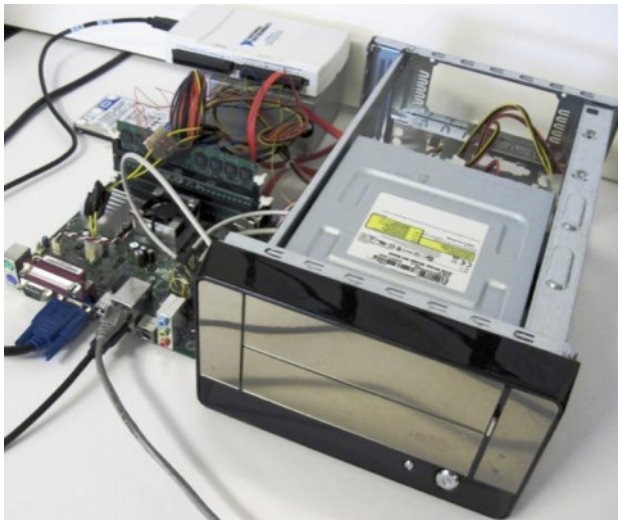

Figure 4.9: LEAP Energy Measurement Tool

Intel processors, Sandy Bridge and later. RAPL is composed of different Machine Specific Registers (MSRs) that provide information about the estimated energy consumption for up to four power planes or domains in a computer. A common client processor (e.g., i3/i5/i7) has one package, while server processors (e.g., Xeon) have two or more packages that contain multiple cores, CPU cache, and/or Graphical Processor Units (GPU). For client processors, querying the RAPL interface allows us to obtain the energy usage of package components.

We decided to use RAPL for several reasons: (1) previous studies have verified that RAPL power estimates are fairly accurate [36, 37, 39, 56, 129, 147], (2) RAPL's interface enables us to seamlessly integrate energy estimates into the SEEDS framework, and (3) RAPL's estimation technique allows us to easily obtain the application's energy usage without incurring the extra costs and complexity introduced by external hardware instrumentation techniques. To query the amount of energy consumed by an application version while running the application's test suite, we adapted the RAPL tool implementation provided by the Mozilla Developer Network [11].

[11] https://developer.mozilla.org/en-US/docs/Mozilla/Performance/tools_power_rapl

Additionally, if a developer would like to estimate the energy usage of an application using other estimation-based strategies, the developer can extend the `EnergyProfiler` class in SEEDS to provide an energy estimation mechanism using tools such as the Intel Power Gadget [12] or the Jalen tool [111].

4.4 Empirical Evaluation

Our evaluation of SEEDS focuses on evaluating the effectiveness of using an instantiation, namely $\text{SEEDS}_{\text{api}}$, on real applications and examining the associated costs. Specifically, we designed our evaluation to answer the following questions:

RQ1—Effectiveness. Is SEEDS effective at automatically optimizing an application with respect to potential code changes?

RQ2—Exploration Capability Can SEEDS be used to effectively explore the search space of the energy impacts of a software engineer's decisions?

RQ3—Cost Can SEEDS provide decision-making support to the software engineer with regard to energy consumption implications at a reasonable cost?

RQ4—Search Space Reduction How do search space reduction strategies impact cost and effectiveness?

RQ5—Metaheuristic-based Search Are GA algorithms an effective approach to improve applications' energy usage by means of introducing code changes?

RQ6—Proxy Measure Can execution time be used as a proxy for the energy usage of an application's API changes?

4.4.1 Experimental Subjects

The primary goal of $\text{SEEDS}_{\text{api}}$ is to help software developers choose implementations of the Collections API to reduce the amount of energy consumed by their Java applications. To suitably evaluate the tool with respect to this goal, we selected nine Java applications that use the Collections API. We selected these programs because

[12] https://software.intel.com/en-us/articles/intel-power-gadget-20

they have been used by many researchers and they are representative of applications that use the JCF. In addition, because $\text{SEEDS}_{\text{api}}$ requires a test suite, we needed to select applications that have an associated test suite.

Table 4.2 describes the nine applications. In the table, the first and second columns, *Application* and *Version*, together identify the application version. The third column, *LoC*, provides the number of lines of code. The fourth and fifth columns, *# Tests* and *Coverage (%)*, reports the number of tests in the associated test suite provided with each subject and the percentage of the statements in the application that are covered by the test suite, respectively. The last column reports the number of possible sites in the application code for the program changes of interest (based on the input parameters).

We obtained the subjects from three different public repositories: (1) Software-artifact Infrastructure Repository (SIR),[13] which provides a variety of open-source projects for empirical software engineering, (2) SourceForge,[14] a popular repository for open-source projects, and (3) Apache Commons,[15] a collection of reusable components.

Table 4.2: Subject Applications.

Application	Version	LoC	# Tests	Coverage (%)	# Change Sites
Barbecue	—	13,610	247	55.9	10
Jdepend	2.9.1	5,865	53	53.2	14
Apache-xml-security	1.0	50,412	175	41.9	15
Joda-Time	2.1	69,225	197	36.6	16
Commons Lang	3.1	100,566	2,046	94.9	47
Commons Beanutils	1.8.3	69,355	1,277	71.3	7
Commons CLI	1.2	8,638	187	96.7	14
Jfreechart	1.0.15	315,787	6,663	67.8	158
Gson	2.2.4	29,119	913	86.6	13

[13] http://sir.unl.edu

[14] https://sourceforge.net

[15] http://commons.apache.org

4.4.2 RQ1: Effectiveness

In our preliminary study (see Section 4.3.1), we observed that switching implementations of the Collections API can improve the energy usage of an application. The goal of our first research question was to determine whether we can achieve the same type of improvements in real applications in a fully automatic manner.

Table 4.3: $\text{SEEDS}_{\text{api}}$ Effectiveness in Improving Energy Usage With Limited Exhaustive Search Strategy.

	% Improvement	
Application	JCF Only	ALL
Barbecue	17*	17*
Jdepend	3*	6*
Apache-xml-security	5†	5†
Joda-Time	8*	9†
Commons Lang	10†	13†
Commons Beanutils	—	—
Commons CLI	2*	2*
Jfreechart	9*	14*
Gson	—	—

* indicates situations where a single concrete change at one change location was most effective.
† indicates situations where a concrete change at every change location was most effective.

To answer this question, we created two improved versions of each of our subjects using $\text{SEEDS}_{\text{api}}$ with the limited exhaustive search strategy. The two improved application versions were created using as input a different set of potential changes, one where $\text{SEEDS}_{\text{api}}$ was allowed to use only Collections implementations from the JCF and one where $\text{SEEDS}_{\text{api}}$ was allowed to use Collections implementations from any of its included libraries. For the cases where $\text{SEEDS}_{\text{api}}$ was able to improve the applications (i.e., it returned a version different than the original), we ran the original and improved versions on the LEAP node 10 times to account for small fluctuations in energy consumption from execution to execution and be able to perform a statistical analysis on the energy impact of application versions. For the statistical analysis, we

used the Kruskal-Wallis test to determine whether there is a statistically significant difference in the amount of energy consumed by the versions. We chose to use the Kruskal-Wallis test because we have one nominal variable (whether or not the change is applied), one measurement value (the amount of energy consumed), and we do not know whether our data are normally distributed. For all of our tests, we chose an alpha (α) of 0.05. For the cases where there was a significant difference in energy consumption, we computed the percentage change in the means of the energy usages of the original and improved versions to determine how effective $\text{SEEDS}_{\text{api}}$ was at improving the energy usage of the applications.

Table 4.3 shows the data that we generated to investigate the effectiveness of $\text{SEEDS}_{\text{api}}$. In the table, the first column, *Application*, shows the name of each subject. The remaining columns show the percentage improvement in energy usage of the optimized version produced by $\text{SEEDS}_{\text{api}}$ when using only implementations provided by the JCF, *JCF only*, and when using the implementation provided by JCF as well as the implementations provided by the other libraries included in the tool (see Section 4.2.3), *ALL*. Note that a dash (—) indicates that $\text{SEEDS}_{\text{api}}$ was unable to improve the application i.e., the search strategy was not able to find an improved version of the application with the provided inputs to $\text{SEEDS}_{\text{api}}$. A $*$ indicates that the optimized version was constructed using only one concrete code change i.e., one single change was applied to one of the change locations in the application. A † indicates that the optimized version was constructed by applying the best individual change at each change location. For instance, for the Jodatime application $\text{SEEDS}_{\text{api}}$ was able to find an improved version composed of one single change when using the implementations from the JCF only as the set of potential changes. When using any of the Collections libraries as the set of potential changes for Jodatime, $\text{SEEDS}_{\text{api}}$ found that the improved version of this application consisted of the application version for which the best Collection implementation, in terms of energy usage, was applied to every change location in Jodatime.

There are several interesting observations that we can make from this data.

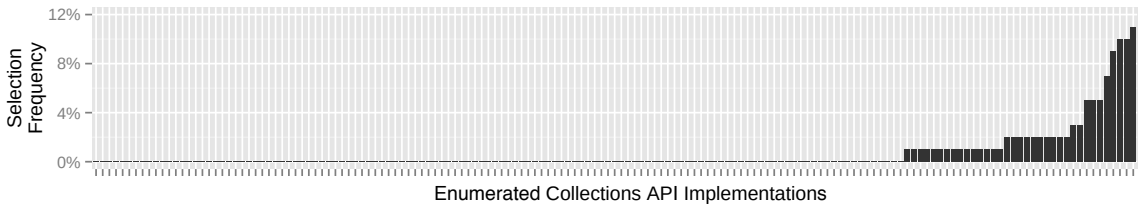

Figure 4.10: Percentage of time each Collection API implementation was selected as the most energy efficient.

First, $\text{SEEDS}_{\text{api}}$ using the limited search strategy was effective at automatically improving the energy usage of our subjects. For all but two applications, (Commons Beanutils and Gson), it was able to decrease energy usage by a statistically significant amount. Moreover, the magnitudes of the changes in energy usage are encouraging as they range from 2 % to 17 % and were accomplished using a simple exhaustive search strategy that only considered changes applied in isolation.

Second, the optimized versions produced by $\text{SEEDS}_{\text{api}}$ include versions that contain only one code change (9 cases), and versions where the most efficient change was made at each location (5 cases). Before running this experiment, we expected the most efficient version to be the version composed of the most efficient change at each change location. The fact that approximately 64 % of the time, the most energy-efficient version contains only a single change at one change location in the whole application, suggests that there can exist complicated interactions among the changes that are canceling out the expected benefits and that more advanced search strategies should attempt to understand and potentially exploit such interactions.

4.4.3 RQ2: Exploration Capability

We posed several questions to examine how well the framework could be used to explore the search space of the energy impacts of a software engineer's decisions to help the software engineer learn more about energy implications of their choices. Specifically, we used the $\text{SEEDS}_{\text{api}}$ to explore the questions:

RQ2a. How does the effectiveness of the energy optimization change with more choices in the set of potential changes?

RQ2b. How often do developers choose the most energy-efficient implementation without knowing the energy efficiency capability of the selection?

RQ2c. How often is each Collections API implementation the most energy efficient?

RQ2a: We can use our previous results from Table 4.3 to answer the question "How does the effectiveness of the energy optimization change with more choices in the set of potential changes?" Comparing columns of Table 4.3 shows that the effectiveness of $\text{SEEDS}_{\text{api}}$ only slightly increases when considering all possible implementations of the Collections API (ALL) rather than just the implementations from the JCF (JCF Only). For 5 of the 9 subjects, adding the additional implementations had no impact on the performance of the $\text{SEEDS}_{\text{api}}$. For the remaining 4 subjects, energy usage was improved, but the magnitude of the improvement was 5 % or less. This was especially surprising as many of the additional implementations are specifically designed to be fast (execute quickly) and compact (use less memory), traits that are commonly thought to be strongly correlated with energy usage [5]. The fact that switching to such implementations does not drastically improve energy usage suggests that the correlation may not be as strong as was previously suspected.

RQ2b: To answer the question of how often developers choose the most energy-efficient option without knowing the energy efficiency capability of the selection, we used $\text{SEEDS}_{\text{api}}$ to determine how often the most efficient implementation choice is different than the implementation used in the original application. In our subjects, there are 208 total locations where an instance of the Collections API is allocated. When only implementations from the JCF are considered, 46 % of the time (96 cases), switching away from the original implementation resulted in a decrease in energy usage. Similarly, when all possible implementations were considered, 62 % of the time (128 cases) switching away from the original implementation improved energy usage. These results motivate the need for SEEDS as they show that developers are often not

choosing the most energy-efficient Collections API implementations.

RQ2c: The final supplemental question we answered is how often each implementation of the Collections API is the most energy efficient. Essentially, we wanted to know if there is a single implementation that is always the most energy efficient. When including all libraries, $\text{SEEDS}_{\text{api}}$ chooses among 157 distinct implementations of the Collections API. Figure 4.10 shows how often each implementation is the most efficient choice. In the figure, the x-axis includes a tick mark for each implementation and is sorted by how often each implementation is the most efficient. The y-axis shows the percentage of times each implementation was most efficient. In our experiments, `CopyOnWriteArrayList`, `ArrayList`, `LinkedList`, `Stack` and `Vector` (all from the JCF) are the implementations that were most frequently the most efficient.

As Figure 4.10 shows, there is not a single implementation that is always the most energy efficient. Moreover, it shows that there are only 63 of the 157 implementations (40%) that were the most energy efficient at least one time. This information further motivates the need for SEEDS. It is unlikely that software engineers would be willing or able to manually investigate tens of possibilities to find the most efficient implementation. This information is also potentially useful for future work in designing better search strategies. While there were 63 implementations that were the best at least once, there were far more implementations (i.e., 94) that were never the most energy efficient. This information could be used to help direct an alternative search strategy.

Table 4.4: Cost to Automatically Improve an Application Using the Limited Exhaustive Search.

Application	Exe. (s)	# Reps.	JCF Only			ALL		
			—Search—	Analysis (hrs)	Cost (hrs)	—Search—	Analysis (hrs)	Cost (hrs)
Barbecue	5	10	63	2	3	242	8	11
Jdepend	4	10	209	5	7	2,004	55	77
Apache-xml-security	124	10	52	46	64	144	125	175
Joda Time	3	10	102	2	3	262	5	7
Commons Lang	90	3	167	32	45	196	37	52
Commons Beanutils	104	3	23	6	8	63	14	19
Commons CLI	2	10	95	2	3	186	3	4
Jfreechart	60	5	908	151	243	5,767	961	1,541
Gson	2	5	57	0.3	0.5	127	0.7	1

4.4.4 RQ3: Cost Using the Limited Exhaustive Search

To answer the question "Can SEEDS provide decision-making support to the software engineer with regard to energy consumption implications at a reasonable cost?", we recorded the times to perform each step of the framework when using the limited exhaustive search strategy for each of the nine subject applications.

Table 4.4 presents the estimated costs, in terms of wall clock time, for improving applications energy consumption using the limited exhaustive search. In the table, the first column, *Application*, shows the name of each application. The second column, *Exe.*, shows the amount of time (in seconds) necessary to execute each application using its test suite once. The third column, *# Reps.*, shows the number of times each changed version was run to gather enough data to compute the percentage difference in the means of the energy usage of the original and changed versions. The fourth column shows the cardinality of the search space (i.e., the number of changes explored by $\text{SEEDS}_{\text{api}}$), —*Search*—. The fifth column, *Analysis*, shows the time (in hours) to analyze the energy usage of the search space. The sixth column shows the cost in hours, *Cost*, when optimizing the applications considering only implementations from the JCF (*JCF Only*). Finally, the seventh, eighth and ninth columns show the *Search Space*, *Analysis* and *Cost*, respectively when considering all implementations included in $\text{SEEDS}_{\text{api}}$ (*ALL*). As shown in the table, the total cost ranges from 0.5 h to 243 h for JCF Only and from 1 h to 1,541 h for ALL.

By far the largest portion of the cost of using $\text{SEEDS}_{\text{api}}$ is collecting and processing the power samples collected when running each changed version. The other parts of the process (i.e., generalizing the application, identifying collections allocation locations in the application, filtering based on optimization parameters, and applying the selected changes) required only a few minutes in total.

Although the overall costs are high, we believe that they are reasonable for two primary reasons. First, optimizing the energy usage of an application is a task that will only be carried out infrequently; most likely as part of the final release process. In this context, even a wait of a few days is likely acceptable as the tool is completely

automated, and could be run in parallel with other pre-release tasks such as integration testing and other forms of quality assurance. Second, the costs of using $\text{SEEDS}_{\text{api}}$ can easily be tailored to fit a software engineer's specific circumstances. As Table 4.4 shows, the overall cost of the technique is determined by 4 factors, the amount of time it takes to execute the application's test suite, the number of repetitions that are run, the time to analyze energy usage data and the size of the application-specific search space and how thoroughly the search strategy explores the search space, all of which are easily controllable by software engineers. Reducing any of these factors will also decrease the cost of using the tool. For example, in our evaluation, we used a limited exhaustive search strategy and ran the entire test suite. Instead, we could have used a non-exhaustive strategy and only executed part of the test suite in order to reduce the cost of using $\text{SEEDS}_{\text{api}}$. In 4.4.5, we show how the costs of SEEDS can be reduced by adapting the search strategy through the selection of either potential changes or locations in the application.

4.4.5 RQ4: Search Space Reduction

The cost of SEEDS is mainly driven by the search space of the potential changes to the application under analysis. Thus, reducing the search space helps to reduce the time SEEDS takes to find an energy improved version of the application. We can reduce the search space by either reducing the number of transformations (e.g., API implementations) or by reducing the number of locations to analyze. Hence, we investigated two space-reduction strategies, one based on the selection of a subset of potential changes, and the other based on the selection of hotspot locations for applying changes in the application. Thus, we formulated two additional research questions:

RQ4a. How does the selection of potential changes impact both effectiveness and cost?

RQ4b. How does the selection of hotspot change locations impact both effectiveness and cost?

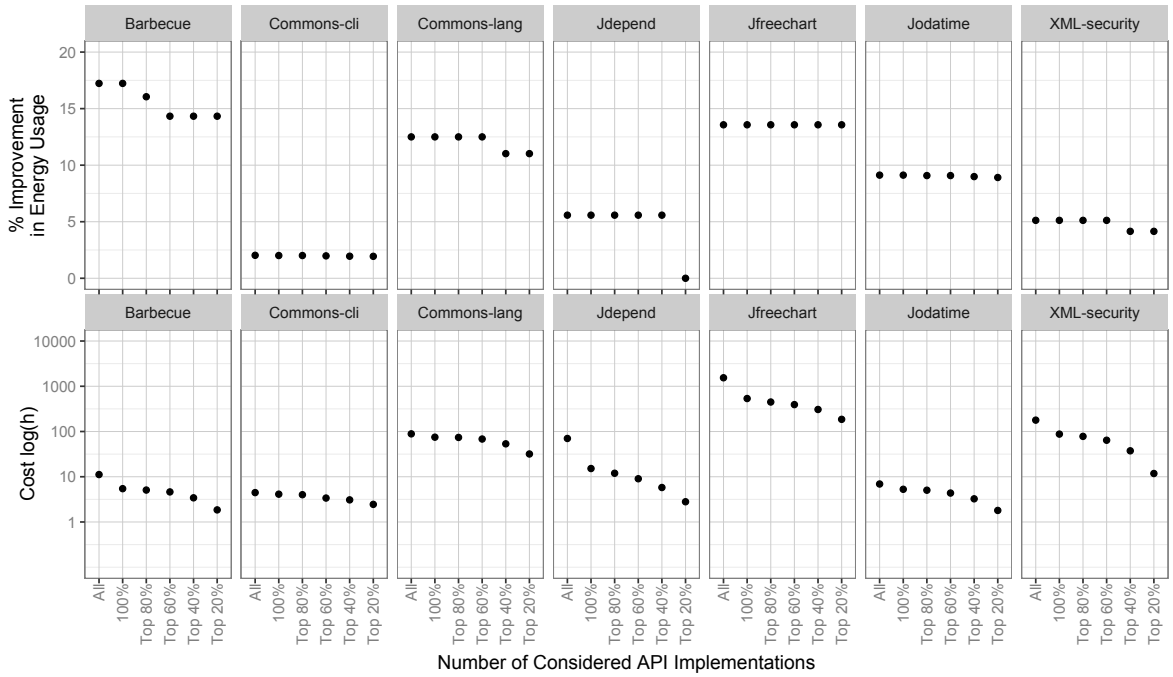

Figure 4.11: Percentage of improvement in Energy Usage and costs by considering different number of API implementations.

RQ4a: The question "How does the selection of potential changes impact both effectiveness and cost?" was answered by investigating the impacts of subsets of potential changes. The subsets of potential changes were selected by considering each implementation's selection frequency i.e., how many times an implementation was selected as an energy-efficient implementation by $\text{SEEDS}_{\text{api}}$, as this indicates that the implementation is more likely to help reduce the energy usage of the application. In section Section 4.4.3, we showed that the number of energy-efficient implementations identified by $\text{SEEDS}_{\text{api}}$ was about 40 % of the implementations associated with all potential changes, i.e., 63 of the 157 available implementations were selected at least once to create an energy-efficient version of an application. The search space considered by SEEDS is greatly reduced by focusing on changes made to create this subset of energy-efficient implementations. Thus, by shrinking the search space under analysis

the cost of the framework as well as the time required to search for an improved application version are reduced. To analyze how subsets of potential changes can help to reduce the cost of $\text{SEEDS}_{\text{api}}$ instead of examining all possible implementations at each location in a subject application, we implemented what we called a "Top X" strategy in $\text{SEEDS}_{\text{api}}$ and explored the "Top X" for X = 20, 40, 60, 80, and 100%. In the "Top X" strategy, only the X % of the top energy-efficient implementations are used as input to the framework.

Figure 4.11 shows the results for both energy usage improvement and costs related with the "Top X" strategy in $\text{SEEDS}_{\text{api}}$. In the figure, we consider the seven subject applications that $\text{SEEDS}_{\text{api}}$ was able to improve when using the limited exhaustive search strategy. In the top row of charts, the y-axis shows the % of improvement in energy usage. In the bottom row of charts, the y-axis shows, in logarithmic scale, the costs of $\text{SEEDS}_{\text{api}}$ to find an improved version of the application. For both rows in Figure 4.11, the x-axis indicates the number of API implementations considered by $\text{SEEDS}_{\text{api}}$ i.e., all implementations (All), all of the 63 energy-efficient implementations (*100%*) or the Top $80 - 20\%$ of the energy-efficient Collections implementations.

In terms of energy usage improvement, the results show that considering fewer implementations maintains the energy savings obtained by $\text{SEEDS}_{\text{api}}$ or reduces them up to only 2%. For instance, for Commons-lang, the 12.50 % of energy-savings obtained when all API implementations are considered remains constant when the Top X strategy selects as few as the Top 40 % of the implementations (i.e., between 63 and 25 implementations are considered to build the search space). Only when the Top X strategy selects only 20 % of the top implementations does the energy savings drop from 12.50 % to 11.02 %, a less than 2 % change in energy savings. A similar pattern can be observed for other applications like Barbecue, Commons-cli, Jodatime and XML-security. For Jfreechart, the energy savings remain constant despite the reduction in the number of selected API implementations in the Top X strategy. For Jdepend, the Top X strategy could be applied when using from 100 % down to 40 % of the top implementations in order to find an improved version of this application, but

there is no energy usage improvement using only the top 20 %.

The results shown in the second row of Figure 4.11 indicate how the Top X strategy helps to reduce the costs of the $\text{SEEDS}_{\text{api}}$. In general, selecting fewer API implementations (i.e., smaller X) reduces the time required by the framework to find an optimal version of an application. For instance, for Barbecue, the cost when taking the Top 20 % API Collections implementations to build the search space helps to reduce the cost to 0.67 hours, which is about $\frac{1}{16}$ of the cost required by $\text{SEEDS}_{\text{api}}$ when using the limited exhaustive strategy.

Overall, reducing the number of selected implementations to the Top 20 % decreases the cost of the framework considerably without dramatically diminishing the energy savings. Only for Jdepend does selecting only Top 20 % Collections implementations restrain the framework to find an energy-efficient version of this application. However, an optimized version for Jdepend can still be found by analyzing only the Top 40 % of the top implementations, for which $\text{SEEDS}_{\text{api}}$ is able to find an optimized version of this application 100 times faster than examining all implementations.

RQ4b: The limited exhaustive search strategy in SEEDS identifies all locations in the subject application where a transformation could be performed by using a given set of potential changes. Only location sites that are covered by the execution of the application's test suite are considered. However, the selection of change locations in the subject applications does not take into account if the locations are hot-spots, i.e., locations that are most frequently executed. By focusing on hot-spot change locations, our aim is twofold: first, examine whether the execution pattern of change locations influences the energy-usage of an application, and second, help to reduce the search space under analysis and therefore minimize the cost of the framework. Hence, our question How does the selection of hot-spot locations impact both effectiveness and cost? was answered by allowing $\text{SEEDS}_{\text{api}}$ to identify hot-spot location sites and then only considering hot-spot change locations when building the search space. The identification of hot-spot locations was done by using the execution count of each location when running the application's test suite. Code locations were sorted in

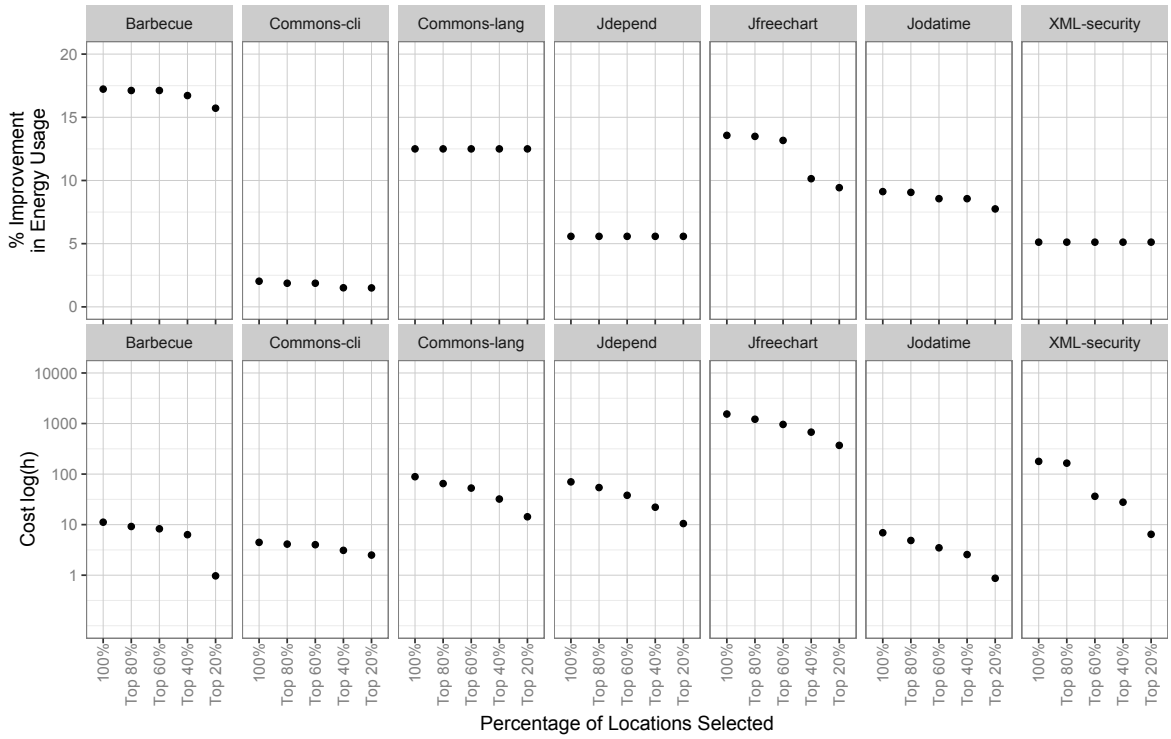

Figure 4.12: Percentage of improvement in Energy Usage and costs of selecting different number of hotspots locations.

descending order by execution count, and the top $Y\%$ were selected to build the search space in $\text{SEEDS}_{\text{api}}$. We varied Y from 20% to 100%.

Similar to Figure 4.11, Figure 4.12 shows the results of improvement in energy usage (top row of charts), and the costs of $\text{SEEDS}_{\text{api}}$ (bottom row of charts) to find an improved version of the application when different percentages of potential change locations (i.e., Y % locations) are considered in the subject applications. In terms of the energy usage improvements, the results show that despite the percentage of hot-spot locations considered, the energy improvement remains constant for Commons-lang, Jdepend and XML-security, and almost invariable for Barbecue, Commons-cli, and Jodatime, i.e., by selecting the Top 20 % of hot-spot locations, the $\text{SEEDS}_{\text{api}}$ is able to find an energy-improved version of the applications. Just for Jfreechart, selecting 40% or less hot-spot locations means a reduction of about 5% in energy savings. In

general, examining only 40 % of hot-spot locations works well for all applications. Furthermore, the costs of the framework (bottom of Figure 4.12) are greatly reduced by applying this strategy. The time required by $\text{SEEDS}_{\text{api}}$ to find an improved version of the subject applications is about half the time than when considering all possible change locations to build the search space.

From these results, we conclude that in general, considering hot-spot strategies not only helps to reduce the cost of the framework but also helps to find a good energy-optimized version of applications that is close to the best optimized version found by analyzing all possible locations. Hence, execution patterns do influence the energy usage of applications, but are not the only factor to consider since for some cases, the best energy-efficient version is not found by considering only this aspect of the applications.

4.4.6 RQ5: Metaheuristic-based Search Strategy

Based on the results presented in Section 4.4.2 for $\text{SEEDS}_{\text{api}}$, we wanted to analyze whether metaheuristic-based search strategies perform better in terms of energy savings and costs compared to the limited exhaustive search strategy. As we described earlier in Section 4.1.2.4, metaheuristic-based search enables the exploration of a more diverse set of application versions belonging to the search space. More specifically, using evolutionary algorithms allows $\text{SEEDS}_{\text{api}}$ to explore application versions that contain combinations of the set of potential changes at different change locations in an application, which increases the portions of the search space that are analyzed and with the advantage of doing a guided exploration to find an energy improved version of an application.

Because metaheuristic-based search strategies allow $\text{SEEDS}_{\text{api}}$ to explore many more application versions than with the limited exhaustive search strategy, we updated our energy profiling mechanism to use the RAPL energy estimation technique, which allows $\text{SEEDS}_{\text{api}}$ to obtain fairly accurate energy estimates for applications versions in less time than with the LEAP profiling technique. Also, RAPL does not require extra

hardware and can be easily integrated into SEEDS to be used online when evaluating the energy usage of a given application version.

To compare the results of the limited exhaustive search strategy with the results of the metaheuristic-based search strategies, the experiments need to be executed under the same conditions. Thus, we present first the results of a reproducibility study of the experiments executed with $\text{SEEDS}_{\text{api}}$ using the RAPL energy estimation technique instead of the LEAP energy profiling technique. Then, we present the empirical evaluation of two metaheuristic-based search strategies implemented in $\text{SEEDS}_{\text{api}}$ using genetic algorithms.

Reproducing $\text{SEEDS}_{\text{api}}$ Results with RAPL

We first conducted a reproducibility study using the RAPL system, presented in Section 4.3.2.5, as the energy profiling technique, which is the same profiling technique used to take the energy samples for the empirical study using the metaheuristic-search approach.

Table 4.5: $\text{SEEDS}_{\text{api}}$ effectiveness comparison using the LEAP and RAPL energy profiling systems.

	% Imp. with LEAP		% Imp. with RAPL	
Application	JCF Only	ALL	JCF Only	ALL
Barbecue	17	17	26	28
Commons CLI	2	2	6	6
Joda-Time	8	9	7	8
Gson	–	–	1	1
Jfreechart	9	14	7	7
Commons-lang	10	13	3	3
Jdepend	3	6	2	2
Commons Beanutils	–	–	1	2
Apache-xml-security	5	5	3	4

Table 4.5 shows the results of the reproducibility study of the results obtained by $\text{SEEDS}_{\text{api}}$, with the limited exhaustive strategy and the RAPL energy profiling

technique, to find energy efficient versions of applications. In this table, the first column *Application*, shows the names of the subject applications; the second column, *% Imp. with LEAP*, presents the percentage of energy savings obtained by the improved versions of the applications found by $\text{SEEDS}_{\text{api}}$ when using the LEAP profiling technique; The third column, *% Imp. with RAPL*, presents the percentage of energy savings obtained by the improved versions of the applications found by $\text{SEEDS}_{\text{api}}$ when using the RAPL profiling technique. For both energy profiling techniques used, the results are divided in two, the *JCF Only* column shows the results for when the exhaustive strategy uses only the implementations from the JCF library, and the *ALL* column shows the results of using the implementations from all the Collection libraries. For instance, for the *Barbecue* application, when using only the JCF library Collections implementations and the LEAP energy profiling technique, we obtained a 17% in energy savings for the improved version found by $\text{SEEDS}_{\text{api}}$, compared to the 26% of energy savings of the solution found by $\text{SEEDS}_{\text{api}}$ when using the RAPL energy profiling method. When using the implementations from *ALL* the Collections libraries for the *Barbecue* application, the experiments with the LEAP and RAPL systems resulted in solutions with 17% and 28% energy savings, respectively.

For some subjects, like *Gson*, the reproducibility study using the RAPL system found a statistically significant solution better than the original version of the application by 1%, which did not happen when using the LEAP system. In other cases, using the RAPL technique in $\text{SEEDS}_{\text{api}}$ yielded solutions with smaller energy savings than when using the LEAP system. For instance, for the *Jodatime* application, when considering changes from the JCF only, the best solution achieved 9% energy savings when using the LEAP system, while with the RAPL system the best solution resulted in 8% energy savings, a difference of 1% in savings between the solutions found using the two different energy profiling techniques. Also, for subjects like *Jfreechart* and *commons-lang*, the magnitude of the energy savings obtained by the solutions found by $\text{SEEDS}_{\text{api}}$ with the RAPL tool were much lower than the magnitude of the reported energy savings when using the LEAP system. For example, for the *Jfreechart*

application when using all the implementations from the collections libraries (i.e., *ALL* column) for making the changes, the reported energy savings when using the LEAP system were 14%, while the energy savings obtained by the solution found when using the LEAP technique is half that value.

The results from this reproducibility study show that there can be large differences between the reported best solutions, and the magnitude of the energy savings, when using two different energy profiling methods and on machines having different hardware characteristics. One explanation to the differences in the reported results can be because of the hardware components considered by the energy estimation technique does not match the hardware components included by the hardware instrumentation energy profiling technique (e.g., making the energy estimation by considering CPU usage only versus considering CPU and Memory). Hence, for comparison purposes, we used the reported results in Table 4.5 as a baseline to compare the results of using the exhaustive search strategy in $\text{SEEDS}_{\text{api}}$ versus the results obtained when using the metaheuristic-based search strategies.

Genetic Algorithms as Search Strategy in $\text{SEEDS}_{\text{api}}$

To answer the question *"Are GA algorithms an effective approach to improve applications' energy usage by means of introducing code changes?"*, we selected two GA algorithms to drive the search for an energy efficient solution in $\text{SEEDS}_{\text{api}}$: One single-objective optimization algorithm i.e., gGA that considers only the energy usage of solutions, and one multi-objective optimization algorithm i.e., NSGA-II that examines both the energy usage and the execution time of solutions in the search space to find an improved version of an application. We used $\text{SEEDS}_{\text{api}}$ with each of the selected GA algorithms to find an improved version of the subject applications. When using the gGA as a search strategy, the output of $\text{SEEDS}_{\text{api}}$ is a solution improving the energy usage only, while using the NSGA-II to drive the search in $\text{SEEDS}_{\text{api}}$ produces solutions that improve both the energy usage and the execution time of the original subject application.

Table 4.6: Comparison between Limited Exhaustive Search and Metaheuristic-based search strategies (gGa, NSGA-II)

Application	% Energy Savings - Limited Exhaustive Search		% Energy Savings - gGA		% Energy Savings - NSGA-II	
	JCF Only	ALL Libs.	JCF Only	ALL Libs.	JCF Only	ALL Libs.
Barbecue	26	28	23	23	28	28
Commons-CLI	6	6	11	12	25	25
Jodatime	7	8	2	2	–	–
Gson	1	1	14	14	5	7
Jfreechart	7	7	13	1	–	3
Commons Lang	3	3	2	2	3	4
Jdepend	2	2	12	18	2	3
C. Beanutils	1	2	2	3	1	2
Apache-xml	3	4	1	1	–	–

Table 4.6 shows the results of using the different search strategies in $\text{SEEDS}_{\text{api}}$ to find energy efficient solutions of the selected subject applications. The first column, *Application*, presents the name of the application, the second, third and fourth columns show the percentage in energy savings obtained with $\text{SEEDS}_{\text{api}}$ when using the limited exhaustive strategy (column *% Energy Savings - Limited Exhaustive Search*), the search strategy driven by the gGA (column *% Energy Savings - gGA*), and when using the search strategy driven by the NSGA-II (*% Energy Savings - NSGA-II*), respectively. For the second to the fourth columns, the results are shown when the implementations used to create the alternative application versions are drawn from the JCF library only, i.e., the *'JCF Only'* column, and when the implementations are drawn from all the Collections libraries, i.e., *'ALL Libs.'* column. A '–' in the table indicates that no better solution than the original application was found by the corresponding search strategy.

From the results shown in this table, we can see that evolutionary algorithms are able to find improved versions of applications in terms of their energy usage. For

instance, for the *Commons-CLI* application, both metaheuristic-based search strategies (i.e., gGA and NSGA-II) found solutions twice to four times more energy efficient than the solutions found for this application when using the limited exhaustive search strategy. For six of the nine applications (e.g., Commons-CLI, Gson, Jfreechart, Commons-lang, Jdepend, and Commons Beanutils), the metaheuristic-based search strategies were able to find better solutions than the solutions found using the limited exhaustive-based search strategy in $\text{SEEDS}_{\text{api}}$. For example, the energy savings obtained by the solution found by $\text{SEEDS}_{\text{api}}$ when using the gGA search strategy is 13% for Jfreechart, compared to a 7% energy savings obtained by the solution found for this application when using the limited exhaustive strategy. Also, for the Gson application, the metaheuristic search using the gGA search strategy found a solution 14 times more energy efficient than the solution found by the limited exhaustive search strategy for this application.

Although the studied metaheuristic-based search strategies using evolutionary algorithms were able to find improved solutions of an application, the results show that evolutionary algorithms can sometimes yield solutions with lower or equal energy savings than solutions found by the limited exhaustive search strategy used in $\text{SEEDS}_{\text{api}}$. For instance, for the *Barbecue* application, the best solutions found by the gGA strategy were $3 - 5\%$ less energy efficient than the solutions found by the limited exhaustive strategy for the same application. Similarly, when using the NSGA-II strategy, $\text{SEEDS}_{\text{api}}$ found a solution with 28% energy savings, which were the same energy savings obtained by the solution found by $\text{SEEDS}_{\text{api}}$ when using the limited exhaustive search strategy. A similar case is presented for the Jodatime subject application, for which only the gGA search was able to find an improved application version with 2% energy savings, not better than the improved application version found by the limited exhaustive search strategy with 8% energy savings. This could be due to the configuration (e.g., starting point, selected genetic operators, and stop criteria) used for the metaheuristic-based search strategies, which in some cases did not direct the search technique to explore portions of the search space where other better solutions

Table 4.7: Costs for Limited Exhaustive Search and Metaheuristic-based search strategies (gGa, NSGA-II)

Application	Cost [hrs] - Limited Exhaustive Search		Cost [hrs] - gGA		Cost[hrs] - NSGA-II	
	JCF Only	ALL Libs.	JCF Only	ALL Libs.	JCF Only	ALL Libs.
Barbecue	0.8	1.6	2.4	5.1	2.3	0.9
Commons-CLI	0.6	0.9	2.0	2.2	1.3	1.2
Jodatime	1.1	4.9	1.5	1.6	–	–
Gson	0.9	2.4	1.0	16.7	2.4	2.3
Jfreechart	16.0	20.2	7.5	41.0	–	5.9
Commons Lang	8.0	26.8	1.2	7.0	10.7	20.1
Jdepend	1.2	1.5	3.4	3.6	2.9	3.1
C. Beanutils	3.3	3.5	2.1	2.3	1.2	1.5
Apache-xml	2.2	2.7	1.6	1.8	–	–

exist for these applications.

In conclusion, both limited exhaustive and metaheristic-based search strategies can be effective strategies to find energy efficient versions of applications within $\text{SEEDS}_{\text{api}}$. Metaheuristic-based search strategies sometimes find solutions with higher energy savings than those obtained with the limited exhaustive search strategy in $\text{SEEDS}_{\text{api}}$, but this is not a general rule. For some applications, using the limited exhaustive search strategy can yield better results in terms of the energy savings obtained by the improved application version. These results motivate the instantiation of multiple search strategies in $\text{SEEDS}_{\text{api}}$ that allow developers to try different search techniques to find an improved energy application version of their applications.

Search Strategies' Costs in $\text{SEEDS}_{\text{api}}$

We also analyzed how the cost of using metaheuristic-based search strategies compare to the cost of using the limited search strategy in $\text{SEEDS}_{\text{api}}$ when using the RAPL interface for profiling the energy usage of applications. Table 4.7 shows, in a similar way to Table 4.6, the costs in terms of the time required for each search strategy

to find an improved version of an application. The first column shows the name of the subject application; the second, third and fourth columns show the time in hours required by $\text{SEEDS}_{\text{api}}$ to obtain an improved version of an application when using the limited exhaustive strategy (column *Cost [hrs] - Limited Exhaustive Search*), the search strategy driven by the gGA (column *Cost [hrs] - gGA*), and when using the search strategy driven by the NSGA-II (*Cost [hrs] - NSGA-II*), respectively. Again, for each strategy, the costs are shown for $\text{SEEDS}_{\text{api}}$ when using the Collections implementations from the JCF only (*'JCF Only'* column), and when using the implementations from all the Collections libraries (*'ALL Libs.'* column). For instance, for the *Barbecue* application, the time required to find an improved application version when using the implementations from the JCF only is 0.8 hrs when using the limited exhaustive search, 2.4 hrs when using the gGA search, and 2.3 hours when using the NSGA-II search. When using the implementations from all the Collections libraries in $\text{SEEDS}_{\text{api}}$, the time required to find an improved version of the *Barbecue* application is 1.6 hours when using the limited exhaustive search, 5.1 hours when using the gGA search, and 0.9 hours when using the NSGA-II search.

From the results presented in Table 4.7, we can see that more often the limited exhaustive search strategy requires less time to find an improved version of an application. For example, for the subjects *Barbecue, Commons-CLI, Jodatime, Gson, and Jdepend*, the limited exhaustive search takes less time to find an improved application version when compared with the time required by the search strategies using the gGA or the NSGA-II. This is because the limited search strategy has a limited set of application versions to explore, while the metaheuristic-based search strategies have to explore a larger portion of the search space looking for the improved application versions. However, in some cases, the metaheuristic-based search strategies are able to find an improved version of an application in about the same or less time than with the limited exhaustive search. This is the case of the *Gson* application, for which the gGA is able to find a solution with better energy savings (i.e., 14% compared to 1% energy savings) than the limited exhaustive search, expending almost the same amount of

time (i.e., one hour). Another example is the *Commons-lang* application, when using the NSGA-II search with the implementations from all the Collections libraries, the time required to find an improved version is about 20.1 hours, compared to 26.8 hours required by the limited exhaustive search for this application. Metaheuristic-based search strategies sometimes need less time to find an improved version of an application due to the way they navigate the search space, which allows them to find better solutions in less time by considering at every generation, the best solutions to find the improved version of an application.

Considering both the amount of energy savings and the cost of each of the search strategies we have analyzed, it seems that the exhaustive limited search strategy can find good energy efficient application versions in a reasonable amount of time, up to one hour for small to medium size applications (*Barbecue, Commons-CLI, Joda-Time, Gson, Jdepend*), and up to 8-16 hours for large applications (i.e., *Jfreechart, Commons-lang, Commons Beanutils, Apache-xml*). Only for three of the nine subject applications (*Commons-CLI*, *Gson*, and *Commons Beanutils*) does the search using the gGA or the NSGA-II lead to better solutions with a cost close to the one needed by the limited exhaustive search. Overall, when the application is larger, more change locations exist where changes can be made to create application versions, therefore increasing the time expended looking for an improved version. Then, for larger applications, the limited exhaustive search strategy seems to be more adequate for $\text{SEEDS}_{\text{api}}$ in terms of the time required to find an improved version of an application and the energy savings obtained. For small to medium size applications, metaheuristic- based search strategies tend to provide a better energy savings to cost ratio than with the limited exhaustive search, and therefore might be preferred. Users can decide which search strategy to use based on their energy and time goals. Although running $\text{SEEDS}_{\text{api}}$ involves some energy costs, these costs are minimal considering that $\text{SEEDS}_{\text{api}}$ needs to be run just once, and compared with the energy savings that can be obtained by finding an energy efficient solution for an application that is anticipated to be run not just once but multiple times and by several users.

4.4.7 RQ6: Proxy Measure

Table 4.8: Correlation between Energy Usage and Execution Time for Subject Applications

Application	τ	p-value
Barbecue	0.185	2.2×10^{-16}
Jdepend	0.288	2.2×10^{-16}
Apache-xml-security	0.092	3.6×10^{-5}
Joda Time	0.353	2.2×10^{-16}
Commons Lang	0.266	2.2×10^{-16}
Commons Beanutils	0.009	0.788
Commons CLI	0.551	2.2×10^{-16}
Jfreechart	0.327	2.2×10^{-16}
Gson	0.322	2.2×10^{-16}

By examining the correlation between energy usage and other variables, such as execution time, we can analyze possible factors in the set of potential changes that influence the way they impact energy usage of applications. A common belief is that energy usage and execution time are directly correlated and therefore execution time can be used as a proxy for the energy consumption of an application, e.g., for test suite minimization [85]. However, the correlation between energy usage and execution time has been shown not to be true in all cases [58]. Hence, we wanted to analyze whether this relation exists for the subjects we included in our evaluation of SEEDS.

We computed Kendall's τ correlation coefficient between the energy usage and execution time for each of the nine subjects described in Table 4.2. For seven of the nine subjects, the correlation coefficient was significantly low ($\tau = [0.009, 0.353]$ with p-value $\ll 0.01$, meaning there is a weak positive relationship between energy usage and execution time. For one of the remaining subjects, although we obtained a low correlation coefficient $\tau = 0.009$, we do not have sufficient evidence to conclude there is a weak correlation for this subject because the result is unlikely to be significant i.e., p-value= 0.79 (see Commons Beanutils in Table 4.8). Finally, for Commons CLI, the

correlation is not strong or weak, (i.e., medium), but the significant $\tau = 0.551$ indicates a positive relationship between energy and execution time for this subject.

4.4.8 Threats to Validity

Threats to construct validity: We evaluated SEEDS by creating one instantiation. It is possible that other instantiations will not lead to improved energy usage of the user's application. For instance, there are other possible search strategies that may provide better energy usage; and although we examined three strategies, they indeed show that SEEDS can result in improved applications that are more energy efficient than the original application. In addition, our study shows that the framework can provide useful information to help understand their energy usage. We also demonstrate that useful instantiations can be created, as choosing a collection implementation is a common decision that is faced by developers, and our results show that indeed SEEDS can automatically make decisions and build optimized versions based on those decisions with regard to energy usage.

Threats to internal validity: Confounding variables include the processes and background tasks running on the machine when the energy usage information was taken. Also, the selected Operating System (OS) and garbage collector (GC) selected, along with the temperature of the room where the experiments took place could have an impact. We minimize this threat by deactivating unnecessary processes and services, like the network and OS update manager of the machine that we used for our experiments; we also controlled the selection of the OS and GC, which were the same for all experiments: XUbuntu 12.04, and the default GC for Java. Finally, the temperature of the room was controlled by keeping the same ambient temperature in the room where the energy measurements were taken.

Threats to external validity: For our evaluation, we selected nine Java applications, used their associated test suites, and chose six libraries as the source of our considered potential choices. It is possible that conclusions drawn from this set may

not generalize to all applications or other languages, libraries, or test suites. To minimize the threat, the applications we considered were selected because they have been used by many researchers and they are representative of applications using the JCF. The test suites are provided by the applications and should thus test typical expected inputs and operations. The libraries all comply with the JCF, are publicly available, and are commonly used. We included libraries that were designed to be fast and compact as well as others designed with a focus on other nonfunctional attributes. Finally, the energy profiling system used in this experiment could be considered a threat to validity. In order to minimize the threat, we used the LEAP monitoring system used by others, which is able to measure the energy of several components (e.g., CPU and memory) and the direct energy of discrete events in kernel and user space systems. In addition, we also used the RAPL interface that has been extensively used and tested by others to estimate the energy usage of applications running on machines using Intel ™technology.

4.5 Related Work

The most closely related work to our SEEDS framework is the one presented in [51], where the design of an autotuning energy model and runtime environment for distributed systems is described. However, the presented model is not evaluated and their implementation requires that developers have knowledge of the hardware components, their interactions, and the energy usage for each different target platform, which is not required by SEEDS.

Autotuning optimization is another related area of work. In autotuning optimization, the goal is to automatically improve the performance of applications. In contrast to common compiler optimizations, autotuning approaches often take into account details about the specific application being optimized and the environment where it will execute. Such approaches have been applied to specific types of software (e.g., computer algebra libraries [152] and high performance computing [122, 148] as well as for general purpose languages and platforms (e.g., [139]).

Of the existing body of autotuning work, Chameleon is most similar to our work. Chameleon is a tool for automatically tuning the collection implementations used by an application [139]. The most significant difference from our work is that Chameleon is focused on runtime performance and memory usage rather than energy efficiency. In addition, Chameleon is a dynamic technique that relies on collecting deep, context-based information about how specific collection instances are used during an execution. In contrast, our approach does not rely on such runtime monitoring as such monitoring is likely to impact the precision of our energy measurements. Unlike for performance tools, the precision of power monitoring tools is insufficient for fine, instruction-level accounting.

Second, there has been some recent work presented by Brownlee et al. and Mahmoud A. Bokhari and Wagner, where approaches based on genetic improvement are used to search and optimize the energy usage of some Collection classes [15] and the Rebound Java library [95]. In [15], Brownlee et al. presented an Object-Oriented Genetic Algorithm (OO-GI) to find an energy-efficient implementation for 6 Collection classes in the Guava and Apache Commons Collections Libraries. Similarly, in [95] a deep parameter optimization algorithm based on genetic algorithms is used to toggle applications' parameters and find versions of an application that improve the original version in terms of its energy usage.

We also examined two genetic algorithms to drive the search for an energy efficient version of an application in $\text{SEEDS}_{\text{api}}$. Our work is different from the existing work using GA in several aspects: (1) the transformations that we carried out are based on implementation changes selected by developers instead of general/random lines of code from a program, (2) the optimizations found by our approach are not necessarily target-specific as those based on assembly code, (3) our study involves real Java programs while previous studies have been applied to isolated classes or to a specific library, and (4) our evaluation is the largest applied to real programs reporting results for six applications with varied code sizes.

Finally, researchers have begun to build on the accurate measurement work

mentioned above to provide source code-level feedback on energy consumption to developers [84]. Although this work is promising, it requires developers to be able to understand the information and manually make any necessary changes. In contrast, SEEDS automatically explores many options without developer involvement.

4.6 Summary

In this chapter, we presented SEEDS, the first known framework for helping software engineers make decisions with regard to energy usage of their applications. Our empirical results show that using such automation can indeed improve the energy usage of real applications without requiring the software engineer to deal with low-level energy profiling tools and analyses. We also presented the design for three instantiations of the framework to analyze and improve the energy of applications by considering decisions commonly made by application developers, such as selecting API implementations, implementing design patterns, or using refactorings for HTTP requests or asynchronous constructs for mobile applications.

We described how $\text{SEEDS}_{\text{api}}$, an instantiation of SEEDS to make decisions about which Collection implementation(s) to use in an application, and showed how this instantiation is able to achieve up to $17 - 28\%$ energy improvement. We analyzed two cost-reduction strategies that helped to shrink the search space explored by $\text{SEEDS}_{\text{api}}$ and to reduce the time of the framework required to find an optimized version of applications. We also presented the analysis of two types of search strategies within $\text{SEEDS}_{\text{api}}$, one using a limited exhaustive search, and two additional based on metaheuristic optimization using GA. Our results show that both kinds of search strategies are able to find improved versions of an application in terms of its energy consumption. Finally, we presented an analysis of the correlation between execution time and energy usage of applications, which show a weak positive correlation overall, showing that execution time is not a good proxy for the energy consumption of applications.

Chapter 5

CONCLUSIONS AND FUTURE WORK

The massive use of mobile and handset devices such as laptops, tablets, and mobile phones, has produced an imminent shift in the type of computer applications and systems that are currently under demand. With the ubiquitous access to computers, mobile devices, and internet, the development of software applications for data centers and resource constrained devices (e.g., laptops, mobile phones) have become more common. With these changes in the demand of software applications usage, energy consumption of applications has emerged as a new important requirement for applications. To aid software developers to be successful at creating energy-efficient applications, it is necessary to first understand how software developers deal with the energy usage of their applications, as well as to develop tools focused on helping developers be successful at making their applications more energy efficient.

In this dissertation, we present a study that shows how practitioners deal with the energy usage of their software applications, as well as our approach to help developers automatically improve applications' energy usage. We developed the SEEDS framework to help developers in their tasks of analyzing the energy impact of the decisions they make in their applications, and improving the energy usage of applications automatically. Specifically, this dissertation's main contributions are:

- A study about practitioners' views and approaches with regard to the energy usage of applications, which helps both the research community and software developers to understand the current state of practice of green software engineering.
- The findings and implications of practitioners' perspectives on green software engineering at the different phases of the software development cycle, which can

be used to guide the design and development of strategies and tools that support practitioners in making their applications more energy efficient.

- A novel framework, SEEDS, that assists developers in analyzing the energy impacts of the decisions they make in their applications, as well as help developers improve applications' energy efficiency through the automatic application of code changes.

- The design of three instantiations of the SEEDS framework that target the selection of API implementations, design patterns, and mobile applications refactorings. Each instantiation enables the analysis of the energy usage and the creation of energy-improved versions of an application.

- An instantiation of SEEDS, called $\text{SEEDS}_{\text{api}}$, that helps developers in the analysis and selection of Collection Implementations that boost the energy savings in their applications.

- The analysis of three different search strategies, based on exhaustive and metaheuristic optimization explorations, to help the SEEDS framework find energy improved versions of applications through the application of distinct combinations of code changes.

5.1 Future Work

5.1.1 Extensions

By creating more instantiations of the SEEDS framework, software developers will be able to analyze and improve the energy usage of their applications using a diverse set of code transformations that are linked to different decisions that they commonly make in their applications. We plan to:

- Leverage the current state of the art in automatic refactorings [72, 75, 87, 88] including design patterns [72, 75] and asynchronous constructs [87] to implement the SEEDS_{dp} and $\text{SEEDS}_{\text{mobile}}$ instantiations.

- Evaluate the effectiveness and cost of other metaheuristic-based search strategies such as Hybrid GA (HGA), Ant Colonies Optimization (ACO), or Particle Swarm Optimization (PSO).

- Focus on code transformations targeted for mobile and web applications, and investigate how well SEEDS performs for applications targeting different hardware platforms such as mobile devices and servers.

- Investigate the effectiveness of SEEDS for applications handling different amounts of data I/O, such as data mining applications, to examine the relationship between the energy consumption of an application and the I/O patterns that belong to different applications' scenarios.

5.1.2 New Directions

In the long term, we will work on investigating some of the open problems associated with improving the energy consumption of applications at different phases of the development cycle, as described in Section 3.2.3. Specifically, we will investigate how changes in energy usage impact other non-functional requirements such as responsiveness, performance, and usability, and we will explore how to develop techniques to determine whether the amount of energy being consumed by an application is adequate for the amount of work being done when executing various applications' scenarios.

Furthermore, we also plan to design and implement an instantiation of SEEDS that allows a developer to analyze how combinations of different types of changes (e.g., refactorings, API implementations, algorithms) can be applied in an application to help practitioners improve the energy usage of their applications. Current instantiations of the SEEDS framework are focused on a single change type (e.g., selecting API implementations, or adopting a design pattern). However, the design of SEEDS enables the adoption of more than one type of transformation to make changes in the application code, which can help to analyze a bigger search space of solutions that

include several different types of changes, with the possibility of finding better energy efficient applications versions.

Bibliography

[1] K. Aggarwal, A. Hindle, and E. Stroulia. Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015.

[2] Tedis Agolli, Lori Pollock, and James Clause. Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft. IEEE Press, 2017.

[3] Jean Alzieu, Hassan Smimite, and Christian Glaize. Improvement of intelligent battery controller: State-of-charge indicator and associated functions. *Journal of Power Sources*, 67(1-2):157–161, 1997.

[4] Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos. Design pattern alternatives: What to do when a gof pattern fails. In *Proceedings of the 17th Panhellenic Conference on Informatics*, PCI, pages 122–127. ACM, 2013.

[5] Nadine Amsel and Bill Tomlinson. Green tracker: A tool for estimating the energy consumption of software. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems: Extended Abstracts*, pages 3337–3342, 2010.

[6] Luca Ardito and Maurizio Morisio. Green it-available data and guidelines for reducing energy consumption in it systems. *Sustainable Computing: Informatics and Systems*, 4(1):24 – 32, 2014.

[7] Rowland Atkinson and John Flint. Snowball sampling. In *The Sage encyclopedia of social science research methods*, pages 1044–1045. Sage Publications, 2004.

[8] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[9] Earl Babbie. *The practice of social research.* Cengage Learning, 13th edition, 2012.

[10] Abhijeet Banerjee. Static analysis driven performance and energy testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 791–794, 2014.

[11] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598, 2014.

[12] Suparna Bhattacharya, Karthick Rajamani, K. Gopinath, and Manish Gupta. Does lean imply green?: A study of the power performance implications of java runtime bloat. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS. ACM, 2012.

[13] P. Bourque and R.E. Fairley, editors. *Guide to the Software Engineering Body of Knowledge, Version 3.0.* IEEE Computer Society, 2014. www.swebok.org.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[15] A. E. I. Brownlee, N. Burles, and J. Swan. Search-based energy optimization of some ubiquitous algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3), 2017.

[16] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO. ACM, 2015.

[17] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, GECCO, pages 1327–1334. ACM, 2015.

[18] Christian Bunse. *On the Impact of Code Obfuscation to Software Energy Consumption*, pages 239–249. Springer International Publishing, 2018.

[19] Christian Bunse, Hagen Höpfner, Suman Roychoudhury, and Essam Mansour. Choosing the "best" sorting algorithm for optimal energy consumption. In *ICSOFT (2)*, pages 199–206, 2009.

[20] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. *Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava*, pages 255–261. Springer International Publishing, 2015.

[21] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 225–236, 2012.

[22] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: How and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 557–566, 2007.

[23] C.-F. Chiasserini and R. R. Rao. Energy efficient battery management. *IEEE Journal on Selected Areas in Communications*, 19(7):1235–1245, 2001.

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

[25] Yi-Fan Chung, Chun-Yu Lin, and Chung-Ta King. Aneprof: Energy profiling for android java virtual machine and applications. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS. IEEE Computer Society, 2011.

[26] Martin Cichy and Jaroslav Jakubik. Design patterns identification using similarity scoring algorithm with weighting score extension. In *Proceedings of the 8th Joint Conference on Knowledge-Based Software Engineering*, pages 465–473. IOS Press, 2008.

[27] Tracy Counts. Running average power limit. Technical report, Intel Open Source, 2012. URL https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl.

[28] Wellisson G. P. da Silva, Lisane Brisolara, Ulisses B. Correa, and Luigi Carro. Evaluation of the impact of code refactoring on embedded software efficiency. In *Proceedings of the 1st Workshop de Sistemas Embarcados*, pages 145–150, 2010.

[29] Barthélémy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 275–284, 2010.

[30] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 189–194, 2010.

[31] J. W. Davidson and S. Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 186–195, 1994.

[32] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. An eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM, pages 1–6. IEEE Computer Society, 2010.

[33] C.R.B. de Souza and D.F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering*, pages 241–250, 2008.

[34] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, 2002.

[35] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 159–169, 2001.

[36] James Demmel and Andrew Gearhart. Instrumenting linear algebra energy consumption via on-chip energy counters. Technical report, EECS Department, University of California, Berkeley, 2012.

[37] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. Initial validation of dram and gpu rapl power measurements. Technical report, University of Maine, 2015. URL http://web.eece.maine.edu/~vweaver/papers/tech_reports/2015_dram_rapl_tr.pdf.

[38] Mian Dong and Lin Zhong. Self-constructive, high-rate energy modeling for battery-powered mobile systems. In *Proceedings of the ACM/USENIX International Conference on Mobile Systems, Applications, and Services*, 2011.

[39] J. Dongarra, H. Ltaief, P. Luszczek, and V. M. Weaver. Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architectures. In *Second International Conference on Cloud and Green Computing*, 2012.

[40] F. Douglis, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

[41] M. Doyle and J. S. Newman. Analysis of capacity-rate data for lithium batteries using simplified models of the discharge process. *Journal of Applied Electrochemistry*, 27(7), 1997.

[42] Johann DREO, Patrick Siarry, Alain Petrowski, and Eric Taillard. *Metaheuristics for hard optimization : methods and case studies*. Springer-Verlag, 2006.

[43] Juan J. Durillo and Antonio J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.

[44] EASED. Energy aware software-engineering and development workshop. http://www.enviroinfo2014.org/index.php/energy-aware-software-engineering-and-development.

[45] N.A. Ernst and G.C. Murphy. Case studies in just-in-time requirements analysis. In *Proceedings of the IEEE 2nd International Workshop on Empirical Requirements Engineering*, pages 25–32, 2012.

[46] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 260–271, 2001.

[47] F. Fotrousi, S.A. Fricker, and M. Fiedler. Quality requirements elicitation based on inquiry of quality-impact relationships. In *Proceedings of the IEEE 22nd International Requirements Engineering Conference*, pages 303–312, 2014.

[48] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[49] Elkin Garcia and Guang R. Gao. Strategies for improving Performance and Energy Efficiency on a Many-core. In *Proceedings of 2013 ACM International Conference on Computer Frontiers (CF 2013)*, pages 9:1–4. ACM, 2013.

[50] B Glaser and A Strauss. The discovery of grounded theory: Strategies for qualitative research. *Aldin Publishing Co.*, 1967.

[51] S. Gotz, Claas Wilke, Matthias Schmidt, Sebastian Cech, and Uwe Assmann. Towards energy auto-tunning. In *Proceedings of 1st Annual International Conference on Green Information Technology*, 2010.

[52] GREENS. International workshop on green and sustainable software. http://greens.cs.vu.nl.

[53] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[54] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, Thirumalesh Bhat, and Syed Emran. Mining energy traces to aid in software development: An empirical case study. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement*, 2014.

[55] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, Mahmut Kandemir, Tao Li, and Lizy Kurian John. Using complete

machine simulation for software power estimation: The SoftWatt approach. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 141–151, 2002.

[56] D. Hackenberg, T. Ilsche, R. Schne, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium on*, pages 194–204, 2013.

[57] Shuai Hao, Ding Li, W. Halfond, and R. Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *First International Workshop on Green and Sustainable Software*, pages 1–7, 2012.

[58] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the International Conference on Software Engineering*, pages 92–101, 2013.

[59] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 217–222, 2003.

[60] A. Hindle. Green mining: Investigating power consumption across versions. In *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[61] Abram Hindle. Green mining: A methodology of relating software change to power consumption. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 78–87, 2012.

[62] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 12–21, 2014.

[63] S.E. Hove and B. Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Proceedings of the 11th IEEE International Symposium Software Metrics*, pages 23–33, 2005.

[64] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 38–48, 2003.

[65] C. Hu, D. A. Jiménez, and U. Kremer. Efficient program power behavior characterization. In *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers*, pages 183–197, 2007.

[66] C. Hu, D. A. Jiménez, and U. Kremer. Combining edge vector and event counter for time-dependent power behavior characterization. In *Transactions on High Performance Embedded Architectures and Compilers II*, pages 85–104. Springer-Verlag, 2009.

[67] P.-K. Huang and S. Ghiasi. Efficient and scalable compiler-directed energy optimization for realtime applications. *ACM Transactions on Design Automation of Electronic Systems*, 12:27:1–27:16, 2008.

[68] ICSE. International conference on software engineering. http://icse-conferences.org.

[69] ICSME. International conference on software maintenance and evolution. http://www.icsme.org.

[70] S. Imamura, K. Oka, Y. Yasui, Y. Inadomi, K. Fujisawa, T. Endo, K. Ueno, K. Fukazawa, N. Hata, Y. Kakibuka, K. Inoue, and T. Ono. Evaluating the impacts of code-level performance tunings on power efficiency. In *IEEE International Conference on Big Data (Big Data)*, pages 362–369, 2016.

[71] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the 2001 Design, Automation and Test in Europe, Conference and Exhibition*, pages 190–196, 2001.

[72] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in java programs. In *Software Engineering Conference, 9th Asia-Pacific*, pages 337–345. IEEE, 2002.

[73] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the International Conference on Software Engineering*, pages 672–681, 2013.

[74] I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. Compiler-directed high-level energy estimation and optimization. *ACM Transactions in Embedded Computing Systems*, 4:819–850, 2005.

[75] J. Kim, D. Batory, D. Dig, and M. Azanza. Improving refactoring speed by 10x. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE, 2016.

[76] Barbara A Kitchenham and Shari L Pfleeger. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer, 2008.

[77] R. Kravets and P. Krishnan. Power management techniques for mobile communication. In *Proceedings of the 4th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 157–168, 1998.

[78] U. Kremer. Low power/energy compiler optimizations. *Low-Power Electronics Design*, pages 2–5, 2005.

[79] W. B. Langdon and M. Harman. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015.

[80] L. Layman, L. Williams, and R.S. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, pages 176–185, 2007.

[81] D. Li, S. Hao, J. Gui, and W. G. J. Halfond. An empirical study of the energy consumption of android applications. In *IEEE International Conference on Software Maintenance and Evolution*, 2014.

[82] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond. Automated energy optimization of http requests for mobile applications. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016.

[83] Ding Li and William G. J. Halfond. An investigation into energy-saving programming practices for Android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 46–53, 2014.

[84] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for Android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89, 2013.

[85] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. Making web applications more energy efficient for OLED smartphones. In *Proceedings of the 36th International Conference on Software Engineering*, pages 527–538, 2014.

[86] X. Li and J. P. Gallagher. A source-level energy optimization framework for mobile applications. In *IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016.

[87] Y. Lin, S. Okur, and D. Dig. Study and refactoring of android asynchronous programming (t). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[88] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016.

[89] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11, 2014.

[90] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE. ACM, 2015.

[91] Andreas Litke, Kostas Zotos, Alexander Chatzigeorgiou, and George Stephanides. Energy consumption analysis of design patterns. In *Proceedings of the International Conference on Machine Learning and Software Engineering*, pages 86–90, 2005.

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

[93] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024, 2014.

[94] Sean Luke. *Essentials of metaheuristics*, volume 2. Lulu Raleigh, 2013.

[95] Brad Alexander Mahmoud A. Bokhari, Bobby R. Bruce and Markus Wagner. Deep parameter optimisation on android smartphones for energy minimisation a tale of woe and a proof-of-concept. In *International Workshop on the Repair and Optimisation of Software using Computational Search*, 2017.

[96] Irene Manotas, Cagri Sahin, James Clause, Lori Pollock, and Kristina Winbladh. Investigating the impacts of web servers on web application energy usage. In *Proceedings of the 2nd International Workshop on Green and Sustainable Software*, pages 16–23, 2013.

[97] MEGSUS. International workshop on measurement and metrics for green and sustainable software. http://www.iwsm-mensura.org/2015/megsus.

[98] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, pages 72–75, 1997.

[99] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: A power/performance/thermal view. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 177–186, 2006.

[100] David Morgan. Snowball sampling. In Lisa M. Given, editor, *The Sage encyclopedia of qualitative research methods*, pages 816–817. Sage Publications, 2008.

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

[102] T. Mudge, T. Austin, and D. Grunwald. The reference manual for the SimPanalyzer version 2.0. http://web.eecs.umich.edu/~panalyzer/.

[103] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings of the 36th International Conference on Software Engineering*, pages 1–11, 2014.

[104] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, March 1993.

[105] E. Musoll. A thermal-friendly load-balancing technique for multi-core processors. In *Proceedings of the 9th International Symposium on Quality Electronic Design*, pages 549–552, 2008.

[106] Stefan Naumann, Markus Dick, Eva Kern, and Timo Johann. The greensoft model: A reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems*, 1(4):294 – 304, 2011.

[107] Nima Nikzad, Octav Chipara, and William G. Griswold. APE: An annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 515–526, 2014.

[108] A. Noureddine and A. Rajan. Optimising energy consumption of design patterns. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[109] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on GreenIT. In *First International Workshop on Green and Sustainable Software*, pages 21–27, 2012.

[110] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. Runtime monitoring of software energy hotspots. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012. ACM, 2012.

[111] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software. *Automated Software Engg.*, 22(3):291–332, 2015.

[112] Adel Noureddine, Syed Islam, and Rabih Bashroush. Jolinar: Analysing the energy footprint of software applications (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA, pages 445–448. ACM, 2016. ISBN 978-1-4503-4390-9.

[113] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[114] C. Pang, A. Hindle, B. Adams, and A. Hassan. What do programmers know about software energy consumption? *Software, IEEE*, 2015.

[115] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, pages 5:1–5:6, 2011.

[116] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42, 2012.

[117] Michael Quinn Patton. *Qualitative evaluation and research methods*. SAGE Publications, inc, 1990.

[118] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 2015.

[119] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, 1998.

[120] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the IpARM microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 96–101, 2000.

[121] N. Pettis, J. Ridenour, and Y.-H. Lu. Automatic run-time selection of power policies for operating systems. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, pages 508–513, 2006.

[122] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 431–444, 2013.

[123] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA. ACM, 2014.

[124] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31, 2014.

[125] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th International Conference on Mobile Computing and Networking*, pages 251–259, 2001.

[126] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting on-line surveys in software engineering. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, pages 80–88, 2003.

[127] Arun Rangasamy, Rahul Nagpal, and Y.N. Srikant. Compiler-directed frequency and voltage scaling for a multiple clock domain microarchitecture. In *Proceedings of the 5th Conference on Computing Frontiers*, pages 209–218, 2008.

[128] Genaína N. Rodrigues, David S. Rosenblum, and Sebastian Uchitel. Sensitivity analysis for a scenario-based reliability prediction model. In *Proceedings of the 2005 Workshop on Architecting Dependable Systems*, pages 1–5, 2005.

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

[130] Cagri Sahin, Furkan Cayci, Irene Lizeth Manotas Gutiérrez, James Clause, Fouad E. Kiamilev, Lori L. Pollock, and Kristina Winbladh. Initial explorations on design pattern energy usage. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 55–61, 2012.

[131] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement*, pages 36:1–36:10, 2014.

[132] Cagri Sahin, Philip Tornquist, Ryan Mckenna, Zachary Pearson, and James Clause. How does code obfuscation impact energy usage? In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 131–140, 2014.

[133] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, pages 2–11, 2002.

[134] Maximilian Schirmer, Sven Bertel, and Jonas Penke. Contexto: Leveraging energy awareness in the development of context-aware applications. In *4th Workshop on Energy Aware Software-Engineering and Development*, pages 131–140, 2014.

[135] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS. ACM, 2014.

[136] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed Java-based systems. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 421–424, 2007.

[137] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Component-level energy consumption estimation for distributed Java-based software systems. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 97–113, 2008.

[138] Chiyoung Seo, George Edwards, Daniel Popescu, Sam Malek, and Nenad Medvidovic. A framework for estimating the energy consumption induced by a distributed system's architectural style. In *Proceedings of the 8th International Workshop on Specification and Verification of Component-based Systems*, SAVCBS. ACM, 2009.

[139] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–418, 2009.

[140] Janice A Singer and Norman G Vinson. Ethical issues in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 28:1171–1180, 2002.

[141] Digvijay Singh, Peter A. H. Peterson, Peter L. Reiher, and William J. Kaiser. The Atom LEAP platform for energy-efficient embedded computing: Architecture, operation, and system implementation. http://lasr.cs.ucla.edu/leap/FrontPage?action=AttachFile&do=get&target=leapwhitepaper.pdf, 2010.

[142] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. Improving developer participation rates in surveys. In *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 89–92, 2013.

[143] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, D. Sarma, and V. Pallipadi. Energy-aware task and interrupt management in Linux. In *Proceedings of the Linux Symposium*, volume 2, 2008.

[144] A Strauss and J Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Inc, 1998.

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

[146] T. J. Watson Libraries for Analysis (WALA), 2015. URL http://wala.sf.net.

[147] Li Tan, Shashank Kothapalli, Longxiang Chen, Omar Hussaini, Ryan Bissiri, and Zizhong Chen. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. *Parallel Computing*, 40(10):559 – 573, 2014.

[148] Ananta Tiwari, Jeffrey K Hollingsworth, Chun Chen, Mary Hall, Chunhua Liao, Daniel J Quinlan, and Jacqueline Chame. Auto-tuning full applications: A case

study. *International Journal of High Performance Computing Applications*, 25 (3):286–294, 2011.

[149] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, pages 384–390, 1994.

[150] Mian Wan, Yuchen Jin, Ding Li, and William G. J. Halfond. Detecting display energy hotspots in Android apps. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10, 2015.

[151] watts up? https://www.wattsupmeters.com/secure/index.php, 2014.

[152] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2): 3–35, 2001.

[153] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[154] Claes Wohlin, Lars Lundberg, and Michael Mattsson. Special issue: Trade-off analysis of software quality attributes. *Software Quality Journal*, 13(4), 2005.

[155] John R. Woodward, Colin G. Johnson, and Alexander E.I. Brownlee. Gp vs gi: If you can't beat them, join them. In *Proceedings of the on Genetic and Evolutionary Computation Conference Companion*, GECCO, pages 1155–1156. ACM, 2016.

[156] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGOPS Operating Systems Review*, 36:123–132, 2002.

[157] C. Zhang, A Hindle, and D.M. German. The impact of user choice on energy consumption. *IEEE Software*, 31(3):69–75, 2014.

[158] Lide Zhang, Mark S. Gordon, Robert P. Dick, Z. Morley Mao, Peter Dinda, and Lei Yang. Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS. ACM, 2012.

[159] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

Appendix

INTERVIEW AND SURVEY QUESTIONS

This appendix presents questions that were included during the interview sessions and an example of the survey that was sent for one of the companies for the the study about Software Developers' perspectives on Green Software Engineering described in Section 3.2 of Chapter 3.

A.1 Interview Questions

- How do you make it easy for developers to write apps that are power efficient?
- How do you work with other teams to bubble up energy performance issues to developers?
- How did you detect when the platform was consuming more power than needed for a scenario?.
- Is there something that got written up based on the experience you had?
- What inputs did you run the applications with? It was just kind of your gut instinct on, okay I see this, and that looks bad, or is it more formal?
- Can you give an example of a situation? So you have access to the source code of these things or are you just using your tracing tool?. Would you take that and sort of manually match it up to what you're seeing on the power meter?
- When you discover a problematic point, how do you proceed usually?

- Would you be able to just look at the event trace without seeing power and say, oh, okay this looks like a pattern I've seen before? Or would you get sort of false positives?
- When you're playing around with these apps, did you have a strategy for doing it?, or are there certain things you found that would more likely lead you into the bad state?, or was it just kind of random see what happens?
- Is there something that an app developer do if they don't have access to the power meter and find these power performance issues?
- How much time compared to writing tests it takes you to do this manual run of the application?
- When you were looking at the apps, was it you would see where you identify bad behavior relative to what the app was doing or was it sort of an absolute scale?
- Do you have any idea of what it means for an app to be efficient? You know what it means for it to be inefficient, but is there a concept of 'this is efficient'? Is there a stopping point where this app couldn't do what it does any more efficiently?
- Would you be able to look at it and say okay, I think you could get x percent more efficient, or you're already close enough to where I think you should be that it's not worth your time to put in more effort?
- When you looked at apps did you look at multiple apps that are sort of fulfilling the same requirements or that are doing the same things i.e., support the same scenarios?
- So if you have three apps that play video, were they all kind of the same overall? If you'd run them for an hour would they consume similar amounts of power or are there ones that are way better?

- Do you see different harbor platforms having an impact on how efficient things can be?
- Is that something you've seen or is that what you suspect?
- Are there things that, so you know, optimizing for CPU usage, optimizing for memory usage, that's always going to be better? Like if I do that I'll always make my power better? Is that something I should focus on as a developer if I want to be more energy efficient?
- Are there other types of tools that you think you would want to have yourself that would make finding opportunities for improving power performance more simple?
- When you went back to the app developers, were they fairly responsive?. Do you think that the app developers care or will care more or less?
- Is there some way that you can make people more aware of what's going on with their phone?. Do you think, sort of testing for energy efficiency is something that you could incorporate into the store?
- How do you define, user profile? Is that based on data and observing people or is just kind of like this is the different ways we intend people to use the application?. Do you actually model scenarios per user profiles in your testing?
- Are you more concerned with an application, kind of battery draw when users aren't actually using it? when it's in background?
- Do you have kind of thresholds or goals that you're trying to hit and how did you come up with those?
- How exactly do you do this? When you say test infrastructure, how do you measure the battery life?

- When you're making changes to an application, do you find that a change you make, makes battery usage better for Windows?, does it make a similar change for the other platforms? Is it always good?
- So is energy performance something that kind of comes later?, like you write your feature and then you see what it does, and if it's bad then you try to make it better?
- Do you feel like once you have designed the feature and then you've constrained the amount of changes you can make in making it better, do you think you would be able to make things a lot better if it was designed up front with more emphasis on making things perform better? Are there rules of thumb that people should keep in mind?
- In a code review, are people able to look at something and say that's going to be horribly inefficient?, or is it something you really need to run to know whether or not it's going to be good or bad?
- Do you have a controlled environment where you do the battery tests?
- Being able to attribute what you see in a trace back up to what caused that is something that would you like to be able to do?
- When you want to try and map the energy usage back up to the application, are you able to map it to a specific point like this function, or this loop, or this line of code, or this feature?
- Do you need the battery trace? Can you just look at the event trace and see what looks weird?
- Are there architectural patterns that are more kind of power friendly?
- Anything else that we haven't touched on that you want to bring up?

A.2 Developer Energy Survey

(1) **Which of the following best describes your title?**

- ○ Software Development Engineer
- ○ Software Development Engineer 2
- ○ Lead Software Development Engineer
- ○ Senior Software Development Engineer
- ○ Principal Software Development Engineer
- ○ Partner Software Development Engineer
- ○ Program Manager
- ○ Software Development Engineer in Test
- ○ Software Development Engineer in Test 2
- ○ Lead Software Development Engineer in Test
- ○ Senior Software Development Engineer in Test
- ○ Principal Software Development Engineer in Test
- ○ Partner Software Development Engineer in Test
- ○ other

Some of the following questions take the form of statements about your beliefs about development or your work. In those cases, please provide an answer that best characterizes how correct those statements are for you.

(2) **I write code that runs on:**

	Never	Rarely	Sometimes	Often	Almost Always
Mobile devices (e.g. tablets or phones)	○	○	○	○	○
Machines in data centers (e.g. Bing web indexing, Azure)	○	○	○	○	○
Traditional PCs (Desktops or Laptops)	○	○	○	○	○
Other	○	○	○	○	○

(3) **Please indicate your agreement with the following statements.**

	Never	Rarely	Sometimes	Often	Almost Always
My applications have goals or requirements about battery life / energy usage	○	○	○	○	○
I'm willing to makes sacrifices in terms of performance, usability, or other aspects of an application for improved battery life / energy usage	○	○	○	○	○
Battery life / energy usage concerns influence how I write new code	○	○	○	○	○

(4) **Concerns about battery life / energy usage impact the design of:**

	Never	Rarely	Sometimes	Often	Almost Always
Individual classes	○	○	○	○	○
Individual modules	○	○	○	○	○
Interaction among parts of an application (e.g., API design, network communication)	○	○	○	○	○
The entire application	○	○	○	○	○
Other	○	○	○	○	○

(5) **When evaluating battery life / energy usage, I consider the following context:**

Usage scenarios	Never	Rarely	Sometimes	Often	Almost Always
Environment (e.g., offline vs. online, wifi vs. 3G)	○	○	○	○	○
Hardware (e.g., graphics card, CPU, radio)	○	○	○	○	○
Other applications or services running concurrently	○	○	○	○	○
Other	○	○	○	○	○

(6) **Please indicate your agreement with the following statements:**

	Strongly disagree	Disagree	Undecided	Agree	Strongly Agree
There are general techniques (design patterns, APIs, architectures, etc.) that lead to good battery life / energy usage	○	○	○	○	○
There are general techniques (anti-patterns, APIs, architectures, etc.) that lead to poor battery life / energy usage	○	○	○	○	○
I have accurate intuitions about the energy efficiency of my code	○	○	○	○	○

(7) **If I was told or decided to improve battery life / energy usage, I could learn how to by:**

	Strongly disagree	Disagree	Undecided	Agree	Strongly Agree
Using tools	○	○	○	○	○
Talking to other developers	○	○	○	○	○
Looking at other code or examples	○	○	○	○	○
Reading papers or other documentation	○	○	○	○	○
Other:	○	○	○	○	○

(8) **Good battery life should be the responsibility of:**

	Strongly disagree	Disagree	Undecided	Agree	Strongly Agree
Applications	○	○	○	○	○
Libraries	○	○	○	○	○
The operating system	○	○	○	○	○
Hardware	○	○	○	○	○
Other:	○	○	○	○	○

(9) **I learn about battery life / energy usage issues in my application from:**

	Never	Rarely	Sometimes	Often	Almost Always
Static analysis	○	○	○	○	○
Profiling	○	○	○	○	○
User feedback (e.g., emails, bug reports, reviews, forum posts)	○	○	○	○	○
Other:	○	○	○	○	○

(10) **I would like to learn about battery life / energy usage issues in my application from:**

	Never	Rarely	Sometimes	Often	Almost Always
Static analysis	○	○	○	○	○
Profiling	○	○	○	○	○
User feedback (e.g., emails, bug reports, reviews, forum posts)	○	○	○	○	○
Other:	○	○	○	○	○

(11) **Compared to other types of performance issues (e.g., execution time, resource usage, responsiveness, etc.), battery life / energy usage issues:**

	Strongly disagree	Disagree	Undecided	Agree	Strongly Agree
Occur more frequently	○	○	○	○	○
Are more difficult to discover	○	○	○	○	○
Are more difficult to diagnose	○	○	○	○	○
Are more difficult to fix	○	○	○	○	○

(12) **Please indicate the relative frequency of the following events in your own work.**

	Never	Rarely	Sometimes	Often	Almost Always
When modifying existing code, I make changes I think will improve battery life / energy usage	○	○	○	○	○
I investigate how the changes I make impact battery life / energy usage	○	○	○	○	○
During code review or other discussions, battery life / energy usage is mentioned	○	○	○	○	○
Documentation (e.g., commit messages, comments, etc.) about changes made to my applications to improve battery life / energy usage exists	○	○	○	○	○

(13) **Do you have any feedback for us regarding this survey or anything that you think would be helpful to know with regard to battery life / energy usage in software development?**