How Do Code Refactorings Affect Energy Usage?*

Cagri Sahin
University of Delaware
United States
cagri@udel.edu

Lori Pollock
University of Delaware
United States
pollock@udel.edu

James Clause
University of Delaware
United States
clause@udel.edu

ABSTRACT

Context: Code refactoring's benefits to understandability, maintainability and extensibility are well known enough that automated support for refactoring is now common in IDEs. However, the decision to apply such transformations is currently performed without regard to the impacts of the refactorings on energy consumption. This is primarily due to a lack of information and tools to provide such relevant information to developers. Unfortunately, concerns about energy efficiency are rapidly becoming a high priority concern in many environments, including embedded systems, laptops, mobile devices, and data centers.

Goal: We aim to address the lack of information about the energy efficiency impacts of code refactorings.

Method: We conducted an empirical study to investigate the energy impacts of 197 applications of 6 commonly-used refactorings.

Results: We found that refactorings can not only impact energy usage but can also increase and decrease the amount of energy used by an application. In addition, we also show that metrics commonly believed to correlate with energy usage are unlikely to be able to fully predict the impact of applying a refactoring.

Conclusion: The results from this and similar studies could be used to augment IDEs to help software developers build more energy efficient software.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

*This work is supported in part by National Science Foundation Grant No. 1216488 and an award from the University of Delaware Research Foundation.

General Terms

Design, Experimentation, Performance, Measurement

Keywords

code refactoring, energy usage, empirical study

1. INTRODUCTION

Software energy efficiency has become an important objective in a broad range of environments where reducing power consumption is a high-priority goal (e.g., embedded systems such as phones and tablets, laptops, and large data centers). Historically, software engineers were unconcerned with energy efficiency; instead they focused on quality attributes such as correctness, performance, reliability, and maintainability. The task of improving energy efficiency was left for compiler writers, operating system designers, and hardware engineers. While the strategy of leaving concerns about energy consumption to the lower-level systems and architecture layers has been successful, recent empirical studies have provided initial evidence that software engineers can play an equally effective role in reducing energy usage through their high-level design and implementation decisions (e.g., [7, 13, 19]).

A major obstacle to developers fulfilling their role in reducing energy consumption is a lack of information about how high-level decisions impact energy consumption. Developers currently do not understand how the choices and tradeoffs they make on a daily basis impact the energy consumption of their software. For example, in preliminary work for this paper, we interviewed 18 developers about how they develop energy-efficient software. A common theme in their answers was that they rarely consider energy efficiency when making high-level decisions, not because they don't care, but because they lack the necessary information and experience to make informed choices.

To combat the lack of knowledge available to developers, researchers have begun to investigate how high-level design and implementation decisions made by software developers impact software energy consumption. These investigations include empirical studies on the impacts of applying design patterns [7, 13, 19], inlining methods [8], and choosing among available components such as web servers [14], sorting algorithm implementations [6], and web browsers [2]. Although these studies have provided encouraging results, they are all preliminary in nature. In addition, they have focused mostly on decisions that are made infrequently rather than the types of choices developers make day-to-day.

One of the most commonly used features in integrated development environments (IDEs) such as Eclipse is the automatic refactoring support that they provide developers [12, 15, 22]. For example, developers can use built-in refactorings to automate common tasks such as extracting code to methods, automatically generating boilerplate code, and introducing indirection. Refactorings typically alter an application to improve its quality in terms of nonfunctional attributes such as readability, understandability, maintainability, etc. (the same properties that developers have historically been focused on). While such changes are often beneficial, they may also have detrimental impacts on the applications's energy consumption.

This paper describes an empirical study of the energy usage implications of applying 6 of the most commonly used refactorings. At a high-level, this study is focused on addressing questions in two major categories: (1) How do different refactorings alter the overall energy consumption of an application?, and (2) How can such effects be characterized, predicted, and presented to the software developer for decision making? To answer our questions, we used Eclipse's refactoring support to create a total of 197 refactored versions of 9 applications. We executed each refactored version on two different platforms multiple times and performed a statistical analysis of the observed impacts of applying refactorings and switching execution platforms on energy usage. In total, we analysed over 350 gigabytes of power profiling information collected from 10,300 executions.

The results of our study show that: (1) all of the refactorings we considered have the potential to impact energy usage, (2) the magnitude of the impacts ranges from decreasing energy usage $\approx 4.6\,\%$ to increasing energy usage by $\approx 7.5\,\%$, (3) refactorings are not consistent in their impacts, and (4) more commonly used and easily collectible information such as execution time and dynamic execution counts are unlikely to be able to accurately predict the energy impacts of applying a refactoring. Moreover, the results provide important initial insights that can be used by software developers when making refactoring decisions and eventually incorporated into the information that an IDE provides along with the refactoring choices.

The remainder of this paper is organized as follows: Section 2 describes the methodology of our study including our subjects and experimental procedure. Section 3 presents and discusses the results of the study including potential threats to its validity. Finally, Sections 4 and 5 discuss related work and present our conclusions and future work.

2. EMPIRICAL STUDY

This section describes the details of our experimental setup including our considered variables, subjects, and data collection protocol. Note that in planning this work, we followed the recommended guidelines for empirical study design [3]. All of our experimental subjects and artifacts, and processed data are publicly available.[1] In addition, all of our raw experimental data is available on request. (The raw data is over 350 gigabytes in size, so it is not feasible to make it publicly available.)

2.1 Experimental Variables

In this study, we considered one dependent variable, the amount of energy consumed by the execution of an application, and two independent variables: (1) the choice of whether or not to apply a refactoring, and (2) the platform where the application executes.

To isolate the impacts of changing our independent variables (applying a refactoring and execution platform) on our dependent variable (energy consumption), it is necessary to control for the effects of several extraneous variables. (e.g., unnecessary changes in the considered application's code and the inputs that are used to drive the application). The remainder of this section describes how we controlled for such extraneous variables.

2.1.1 *Controlling for Extraneous Changes in an Application's Code*

In many cases, refactorings are not formally specified. Because of this, different people, or even different tools, may use the same name to refer to different sequences of code changes. This flexibility in nomenclature can be a potential source of bias and a potential source of confusion in interpreting the results of the study. If we compared the impact of refactorings that were inconsistently applied, we would essentially be comparing different refactorings. Similarly, if a developer would apply a substantially different set of code edits that happen to share the same name as one of the refactorings that we considered, the results that they observe could be drastically different than what we observed.

In order to avoid these potential problems, we must ensure that all refactorings are applied in a consistent, repeatable, and well documented manner. To accomplish this, we relied on the automated refactoring support provided by the Eclipse IDE version 3.7.2 (Indigo). By using the tools provided by Eclipse, we are ensuring that the changes we are making to our considered applications are the same changes that a developer would apply if they applied the same refactoring using the same tool.

2.1.2 *Controlling for Inconsistencies in Driving an Application*

In general, applications are interactive. They accept input, perform some computation, and generate a response. In our experiments, this interactive nature can introduce a potential source of bias as it is difficult to manually reproduce a given execution exactly. For example, a user can often repeatedly perform the same sequence of actions (e.g., enter text into a form or click a button), but can not 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 changing our independent variables, but rather differences in how the application is driven. In order to prevent such bias it is necessary to be able to deterministically reproduce a given sequence of actions with great fidelity; unit testing frameworks provide this capability.

Unit testing frameworks (e.g., JUnit[2]) are commonly used as part of the software development process. They allow developers to encode how an application should respond when given certain inputs. Such descriptions are then executed and checked by an automated driver component. 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

[1] `https://bitbucket.org/udse/refactoring-study`

[2] `http://junit.org`

variations in energy consumption are more likely to be the result of changing an independent variable rather than inconsistencies in driving the application.

2.1.3 Considered Applications

In our study, we investigated the impacts of applying refactorings on 9 Java applications. The specific applications we selected are described in Table 1. The first two columns, *Name* and *Version*, indicate the name and version number of each application, respectively. A blank version number indicates that the corresponding application has only a single version. The third and fourth columns, (*# Classes* and *# Methods*), provide the number of classes and the number of methods in each application, respectively. The number of lines of code is reported in the fifth column, *LoC*, and the number of JUnit tests provided with each application is shown in the sixth column, *# Tests*. The final column, *% Coverage*, is the percentage of statements covered by the test suite. For example, version 1.2 of Commons CLI consists of 21 classes, 192 methods, 4,739 lines of code, and comes with 187 tests that cover 96 % of the application's statements.

We chose these applications for several reasons. First, they represent a variety of application domains. For example, Commons CLI is a library for processing command-line options, Commons IO is a library for performing various input/output operations, and Joda-Time is a library for handling dates and times. By selecting applications from varied domains, we can improve the generalizability of our results. Second, the applications vary in size. For example, Commons Math has over 100,000 lines of code, while Sudoku only has 497 lines of code. Refactorings are not only applied to large, well established projects. They are also used in the context of new or relatively small projects. Again, selecting applications of various sizes can improve the generalizability of our results. Finally, they come with extensive test suites. As we mentioned in Section 2.1, we are using JUnit tests to drive the applications. We believe that extensive tests are more likely to cover large portions of the application's functionality and to drive the applications in ways that match their expected behavior. In addition to fulfilling our requirements for driving the applications, the unit tests also helped guide the choice of where to apply refactorings in each application (see Section 2.3).

2.1.4 Considered Refactorings

To select the refactorings we considered in our study, we first examined all of the refactorings provided by the Eclipse IDE. We filtered this initial list based on two criteria: (1) the refactorings we select should be commonly used, and (2) applying the refactorings should make some structural change to the application.

To determine how often a specific refactoring is applied by developers, we examined the publicly available data gathered by the Eclipse Usage Data Collector (UDC).[3] From this data, we identified the most commonly used editing commands (excluding navigation commands and formatting, organizing, and boilerplate generation actions). We then filtered the remaining refactorings and eliminated ones that make no structural changes. For example, although Rename Variable and Rename Method are among the most commonly used commands, the changes that they make are not evident in the application's compiled bytecode. As such, they have no possibility of altering the amount of energy consumed by the application.

Finally, we sorted the remaining refactorings by how often they can be applied to our Java applications. Some refactorings (e.g., Convert Anonymous Inner Class to Nested Class) can only be applied in very specific circumstances. Since we were interested in identifying general trends about how refactorings impact energy consumption, refactorings that can only provide a single data point are not very useful. To estimate the number of times a refactoring could be applied, we manually examined the code of each application and searched for locations that satisfy the necessary conditions for each refactoring.

As our final set of refactorings to apply, we selected the following 6 refactorings (listed in alphabetical order):

Convert Local Variable to Field: Creates a new field by turning a local variable into a field

Extract Local Variable: Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable

Extract Method: Creates a new method containing the currently selected statement or expression and replaces the selection with a reference to the new method

Introduce Indirection: Creates a static method that can be used to indirectly delegate to the selected method

Inline Method: Copies the body of a callee method into the body of a caller method

Introduce Parameter Object: Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduced parameter

These refactorings all fulfill our requirements: they are commonly used and they cause structural changes that are reflected in an application's compiled bytecode.

2.1.5 Considered Platforms

In our study, we executed the original and refactored versions of each application on versions 7u25 (JVM 7) and 6b27 (JVM 6) of the OpenJDK Java Runtime Environment (JRE). We chose these versions because they are the versions most commonly used in practice.

Although, from a programmer's perspective, there may not appear to be many changes between JVM 6 and JVM 7, there are indeed a significant number of differences. For example, the performance of JVM 7 was improved by techniques such as Tiered Compilation, Compressed Oops (ordinary object pointers), Zero-Based Compressed Oops, and Advanced escape analysis. In addition to improving performance, JVM 7 changes also affected how internal strings are stored (they moved from being part of the permanent generation of the Java heap to the main part of the Java heap), the verifier, and the default garbage collector. All of these changes have the potential to interact with the modifications made by refactorings. Thus, investigating how the the refactorings applied on different underlying platforms impact energy consumption can give valuable information to developers depending on where their application will be deployed.

2.2 Measurement

To measure the amount of energy consumed by executing a subject, we used a Low Power Energy Aware Process-

[3]http://www.eclipse.org/org/usagedata

Table 1: Java applications.

Name	Version	# Classes	# Methods	LoC	# Tests	% Coverage
Commons Beanutils	1.8.3	118	1,199	31,538	1,514	63
Commons CLI	1.2	21	192	4,739	187	96
Commons Collections	3.2.1	412	3,796	63,852	39,143	81
Commons IO	2.4	108	1,069	25,663	966	89
Commons Lang	3.1	147	2,219	55,626	2,047	94
Commons Math	3.0	666	4,974	135,796	3,451	83
Joda-Convert	1.2	10	65	1,317	105	93
Joda-Time	2.1	226	3,731	67,590	11,663	88
Sudoku	—	4	57	497	25	81

ing (LEAP) node [20]. Our LEAP node is an x86 platform based on an Intel Atom motherboard (D945GCLF2). It is currently configured with 1 GB of DDR2 memory, 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 data acquisition (DAQ) card (National Instruments USB-6215) that continually samples the amount of power consumed by the component at a rate of 10 kHz ($\approx$ 10,000 samples per second). The LEAP node also provides running applications with the ability to trigger a synchronization signal. This allows for synchronizing the power samples with the portions of the execution that are of interest. We use this functionality to identify precisely when the test suites start and stop executing.

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 that only contains timestamps. Because power samples are collected by hardware instrumentation, it is necessary to synchronize them with the application execution to identify, in terms of the application, when a specific power sample was taken. It is possible to account for costs of collecting synchronization information by profiling the energy cost of recording such information and subtracting it from the reported energy numbers. However, because we are concerned with energy consumption relative to a base line (i.e., before and after applying a refactoring) rather than absolute numbers and because the energy cost of recording synchronization information is essentially constant, we have removed this step.

2.3 Procedure

Figure 2 shows, at a high-level, the procedure we followed in our study, divided into three main steps: *Subject Creation*, *Data Collection*, and *Post Processing*. The remainder of this section describes these 3 steps in detail.

2.3.1 Subject Creation

The first step in our procedure is to create our set of experimental subjects. Because we are interested in the impacts of applying a refactoring to an application, our experimental subjects are versions of our considered applications with a refactoring applied. To create the necessary refactored versions, we carried out the following sequence of actions.

```
public List<Box> getPeers(Puzzle puzzle) {
  ArrayList<Box> peers = new ArrayList<Box>();
  addUnique(peers, getPeersInSameRow(puzzle));
  addUnique(peers, getPeersInSameColumn(puzzle));
  addUnique(peers, getPeersInSameSubSquare(puzzle));
  return peers;
}

private void addUnique(ArrayList<Box> peers,
                       List<Box> peersInSameRow) {
  for (Box peer : peersInSameRow)
    if (!peers.contains(peer))
      peers.add(peer);
}
```

(a) **Original**

```
public List<Box> getPeers(Puzzle puzzle) {
  ArrayList<Box> peers = new ArrayList<Box>();
  for (Box peer : getPeersInSameRow(puzzle))
    if (!peers.contains(peer))
      peers.add(peer);
  for (Box peer : getPeersInSameColumn(puzzle))
    if (!peers.contains(peer))
      peers.add(peer);
  for (Box peer : getPeersInSameSubSquare(puzzle))
    if (!peers.contains(peer))
      peers.add(peer);
  return peers;
}
```

(b) **Refactored**

Figure 1: Applying the Inline Method refactoring to `addUnique`.

First, for each considered application, we used Atlassian's Clover coverage tool (version 3.1.11)[4] to identify the portions of the application that are covered by its test suite. This coverage information serves as a filter to prevent applying a refactoring in a segment of the application that is not executed by the test suite. The impacts of refactorings in such areas would be unobservable because the code would not be executed.

Next, we identified a set of suitable locations where each refactoring could be performed. For each refactoring, we manually examined the covered portions of each application and searched for locations where the preconditions necessary for applying a refactoring are satisfied. We then attempted to apply the refactorings at the selected locations to create 4 different refactored versions of each application. Basically, a refactored version was created by applying a refactoring at a selected location. Sometimes less than 4 versions, 4 is the maximum number of location where refactoring could be apply for the smallest application, could be created because there exists no possible locations for that refactoring.

To actually apply the refactorings, as mentioned previously, we used Eclipse's built-in refactoring tools. Figure 1 shows, for a code excerpt from Sudoku, (a) the original and (b) refactored versions of the code when applying Inline Method to the `getPeers` method. Note that not every

[4]`https://www.atlassian.com/software/clover/overview`

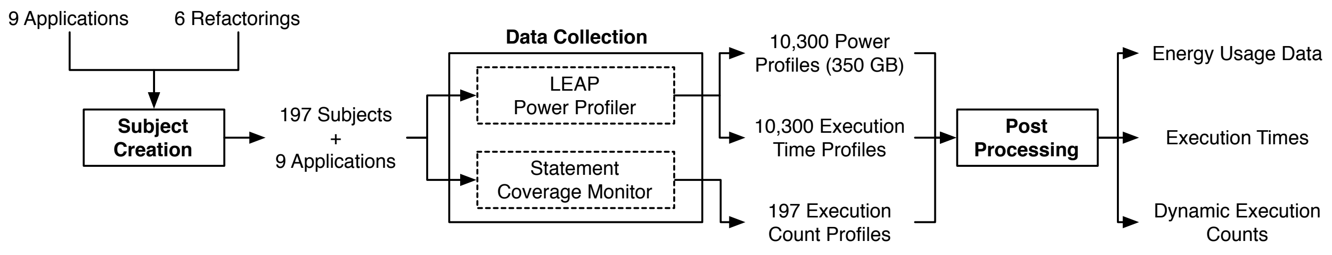

Figure 2: Subject creation and data collection procedure.

refactoring attempt was successful; in several cases, Eclipse was unable to perform a refactoring due to an internal error.

When applying the refactorings, Eclipse provides configuration options for all of our considered refactorings. The options and the parameter values for those options that we used for each refactoring are listed below:

Convert Local Variable to Field: The new field created by the refactoring can be made "public", "protected", or "private". We chose to make it public. Also, we chose to initialize the new field at its declaration location instead of in the current method, when it was possible.

Extract Local Variable: All occurrences of the selected expression can be replaced by a reference to the newly created variable, or only the selected expression can be replaced. We chose to replace all occurrences.

Extract Method: The extracted method can be created with "public", "protected", or "private" protection. We chose to make the extracted method public.

Introduce Indirection: Either all method invocations can be redirected to the newly created static method, or only the selected method invocation can be redirected. We chose to redirect all method invocations.

Inline Method: The method to be inlined can be inlined into every caller method or only into the selected caller method. We chose to inline it into every caller method if it is applicable.

Introduce Parameter Object: The new parameter object class can be a top-level class or nested within the current class. We chose to create the new class at the top level. In addition, the signature of the existing method can be changed, or it can be modified to be a proxy method (i.e., the method simply packages its arguments in an instance of the new parameter object class and passes along the new object.) We chose to modify the method rather than keep it as a delegate.

In total, we created 197 subjects. Five of the refactorings, Convert Local Variable to Field, Extract Method, Introduce Indirection, Inline Method, and Introduce Parameter Object, were successfully applied 36 times each, 4 times in each of our 9 applications. The remaining refactoring, Extract Local Variable, was only successfully applied 17 times, 1 time in Commons Collections, 2 times in Commons IO, 3 times in Commons Lang and Joda-Time, and 4 times in Commons Beanutils and Commons Math.

2.3.2 Data Collection

To answer our research questions, we collected three different types of data: (1) power usage data, (2) execution times, and (3) dynamic execution counts. To collect this data, we first created a set of Apache Ant build files for executing each experimental subject using its test suite. Using an Ant file allows us to execute the subjects with a single command and from the command line. Both of these properties are important as they help reduce noise when executing the subjects.

Power consumption. To collect power usage data, we executed each subject on the LEAP node 25 times using JVM 6 and 25 times using JVM 7. To further reduce the possibility of noise, we disconnected the LEAP node from the network, booted into single user mode, and terminated all unnecessary applications and processes. Although we eliminated many possible sources of noise by carefully configuring the LEAP machine, small fluctuations in energy consumption from execution to execution are still possible. Using multiple runs (i.e., 25) allows us to perform a statistical analysis on the impact of refactorings that takes into account the possibility of such fluctuations.

While each subject was executing, we sampled the power usage of the entire system. In total, we ran 10,300 executions—(197 subjects + 9 original applications) × 25 repetitions × 2 platforms—which took over 15 days worth of CPU time and resulted in over 350 gigabytes of raw power usage data.

Execution Time. To collect accurate execution times, we again used the LEAP node. As we mentioned in Section 2.2, the LEAP node records synchronization information. This synchronization information includes timestamps that correspond to the start and end of the execution. By using this information, we can calculate the total execution time of each execution. Again, this process resulted in 10,300 data points.

Dynamic Execution Counts. The final type of data we collected was how many times each location where the refactorings were applied was executed by the test suite. To calculate this information, we again used Atlassian's Clover coverage tool but this time we recorded how many times each statement in each application was executed rather than only recording whether each statement was executed. Note that to collect this information, we only needed to consider one execution of the original, unmodified version of each application. The execution counts for each of the 197 subjects can be calculated from just this coverage information.

2.3.3 Post Processing

The final step in our procedure is to post process all of the collected data. Dynamic execution counts and execution times are usable in their current form, but the power consumption data needs to be synchronized, filtered, and converted to a useable form.

We post-processed the raw power usage data to calculate the total energy usage of each execution. Then, we grouped the collected data by application, applied refactoring, and platform used for the execution. Because of the large size of the power profiles, post processing this data took an additional 25 days worth of time.

3. DATA ANALYSIS AND DISCUSSION

We refined our overall question of whether or not applying a refactoring can impact the energy usage of an application into the following specific research questions:

RQ1—Impact. Do refactorings impact the energy usage of an application? If so, how?
RQ2—Consistency. Are the effects of applying a refactoring consistent across applications and across platforms?
RQ3—Predictability. Is it possible to predict the impact on energy usage of applying a refactoring by examining data that is more easily accessible?

The remainder of this section discusses the results of our study in terms of these research questions.

3.1 RQ1—Impact

To gather the data necessary to answer our first research question, we performed a Mann-Whitney-Wilcoxon test to determine whether the difference between the amount of energy consumed by the original version and refactored version of each subject is statistically significant. We chose to use the Mann-Whitney-Wilcoxon test because we have one nominal variable (whether or not the a refactoring is applied), one measurement value (the amount of energy consumed), and we do not know whether our data are normally distributed. We chose an alpha (α) of 0.05 and used R version 2.14.1's implementation of the test (i.e., `wilcox.test`). We also used Vargha and DeLaney's $\hat{A}_{12}$ measure to calculate the size of each effect [21].

Of the 394 tests that we conducted (197 for each platform), 109 ($\approx 28\,\%$) indicated a statistically significant difference in energy usage between the original and refactored versions. Note that, according to the guidelines for interpreting $\hat{A}_{12}$ [21], the size of the effect for all of these cases is either "large" (greater than 0.71 or less than 0.29) or "medium" (greater than 0.64 or less than 0.36). This result demonstrates that, although refactorings do not always affect energy usage, it is possible for developers to impact the energy consumption of their applications by performing refactorings. Since refactorings are common, even if not every refactoring performed by a developer impacts energy consumption, developers are likely to perform at least a few refactorings that do indeed impact energy usage.

To gain additional insight into how the refactorings impact energy usage, we investigated how many times each considered refactoring had a statistically significant impact on energy usage. This information is shown in the first part of Table 2. In the table, the first column, *Refactoring*, lists each of our considered refactorings. The second column, *# Subjects*, shows the number of subjects that were created by applying the refactoring. The third and sixth columns, *Total*, show the number of times each refactoring caused a statistically significant difference in energy usage when a subject was executed using JVM 6 and JVM 7, respectively. As this data shows, the 109 cases where a difference occurs are split relatively equally over the 6 refactorings with Convert Local Variable to Field and Introduce Parameter Object making a difference most often (13 out of 36 for JVM 6 and 12 out of 36 for JVM 7) and Extract Local Variable making a difference least often (3 out of 17 for JVM 6 and 0 out of 17 for JVM 7). Most importantly, the data reveals that every refactoring has the potential to impact energy usage.

The next dimension that we investigated was how frequently each refactoring increased energy usage and how frequently each refactoring decreased energy usage. To answer this question for the cases where there is a significant difference, we manually examined our data and determined whether the energy usage of the refactored version was more or less than the original version. The results of the investigation are also shown in Table 2. In the table, columns four and seven, *# NI*, show the number of times each refactoring had a negative impact (i.e., increased energy usage) for JVM 6 and JVM 7 and columns five and eight, *# PI* show the number of times each refactoring had a positive impact (i.e., decrease energy usage), again, for JVM 6 and JVM 7. For example, Extract Method increased energy usage 8 times and decreased energy usage 2 times on JVM 6. Similarly to how every refactoring has the potential to impact energy usage, each refactoring, with the exception of Extract Local Variable, both increased and decreased energy usage.

Finally, we investigated the magnitude of the differences caused by the refactorings. To determine the magnitude of the differences, we again focused on the cases where there is a significant difference. We calculated the percentage change in the means of the energy usages of the original and refactored versions. The results of these computations are shown in Figure 3. This figure is composed of 6 subfigures, one for each refactoring. In each subfigure, the x-axis shows each of our 9 applications, and the y-axis shows the percentage improvement in energy usage between the original and refactored versions. For example, 2 of the times Inline Method was applied in Commons BeanUtils resulted in a percentage change of $\approx 0.75\,\%$, whereas no application of Extract Local Variable in Commons Lang resulted in a significant change. Note that in this figure, positive values on the y-axis indicate that energy usage improved (i.e., decreased) and negative values indicate that energy usage degraded (i.e., increased). Also note that the points have been "jittered" along the x-axis, to make it more obvious when several points overlap. Finally, the shape of each dot indicates the platform that was used to execute the subject: a ● indicates that JVM 6 was used, and a ▲ indicates that JVM 7 was used.

As Figure 3 shows, the percentage change in energy usage ranges from $-7.50\,\%$ to $4.54\,\%$. In our prior work on investigating the energy impact of applying design patterns [19], we found that applying a design pattern could increase or decrease energy usage by several hundred percent which is significantly larger than the impacts we observe here. However, applying a design pattern is likely to change more of an application's code than applying a refactoring so, intuitively, it makes sense that they can have a larger impact. Another factor to consider is how frequently each type of change is made. Design patterns are usually applied once, while refactorings are applied repeatedly. Therefore, it may be possible that the cumulative effect of applying several refactorings could approach the impacts we observed for design patterns.

Based on our investigations of the energy usage impacts of refactorings, we have found that: (1) it is possible that applying a refactoring can significantly impact the energy usage of an application, (2) all of our considered refactorings can both increase and decrease energy consumption,

Table 2: Number of times each refactoring causes a statistically significant difference in energy usage ($\alpha \leq 0.05$).

		JVM 6			JVM 7		
Refactoring	# Subjects	Total	# NI	# PI	Total	# NI	# PI
Convert Local Variable to Field	36	13	5	8	12	3	9
Extract Local Variable	17	3	0	3	0	0	0
Extract Method	36	10	8	2	9	7	2
Inline Method	36	9	4	5	7	4	3
Introduce Indirection	36	12	9	3	9	8	1
Introduce Parameter Object	36	13	4	9	12	4	8
Total	197	60	30	30	49	26	23

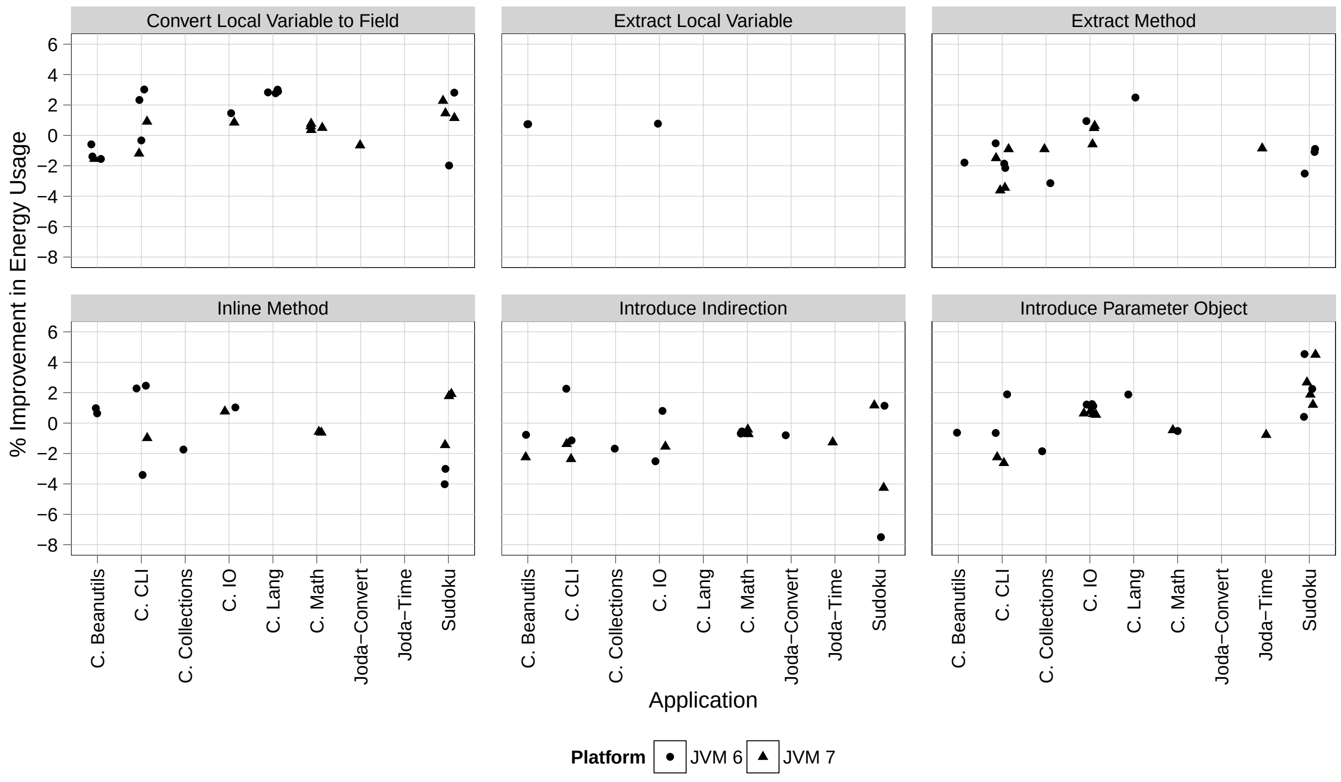

Figure 3: Impacts on energy usage of applying refactorings ($\alpha \leq 0.05$).

except Extract Local Variable, (3) the likelihood of causing an increase or decrease is approximately the same, and (4) both beneficial and negative impacts have similar maximum percentage change values.

3.2 RQ2—Consistency

The goal of our second research question is to determine whether refactorings are consistent in how they impact energy usage: (1) within an application, (2) across applications, (3) within a platform, and (4) across platforms. To answer these questions, we again used the data presented in Table 2 and Figure 3.

For all 4 questions, it appears that the refactorings are not consistent in their impacts. As Figure 3 indicates, the refactorings are not consistent within each application. With the exception of Extract Local Variable, for each refactoring, there is at least one case where the refactoring that causes energy usage to increase and one case that causes energy usage to decrease within an application. For example, applying Convert Local Variable to Field to Commons CLI caused the energy usage to increase in to cases and to decrease in three cases. As Figure 3 shows, the impacts of the refactorings are not consistent across applications. In many cases, a refactoring that causes a significant difference several times in one application never causes a significant difference in another application. For example, Convert Local Variable to Field causes a significant decrease in the energy usage of Commons Math but does not cause a significant difference in Commons Collections or Joda-Time. Moreover, a refactoring may decrease energy usage in one application but increase it in another application. Similarly, refactorings are not consistent within platforms. There are cases where, when run on the same platform, refactorings both increase and decrease the energy usage of different applications. For example, when run on JVM 7, Inline Method decreases the energy usage of Commons IO but increases the energy usage of Commons Math. Finally, the refactorings are not consistent across platforms. Again, there are cases where applying

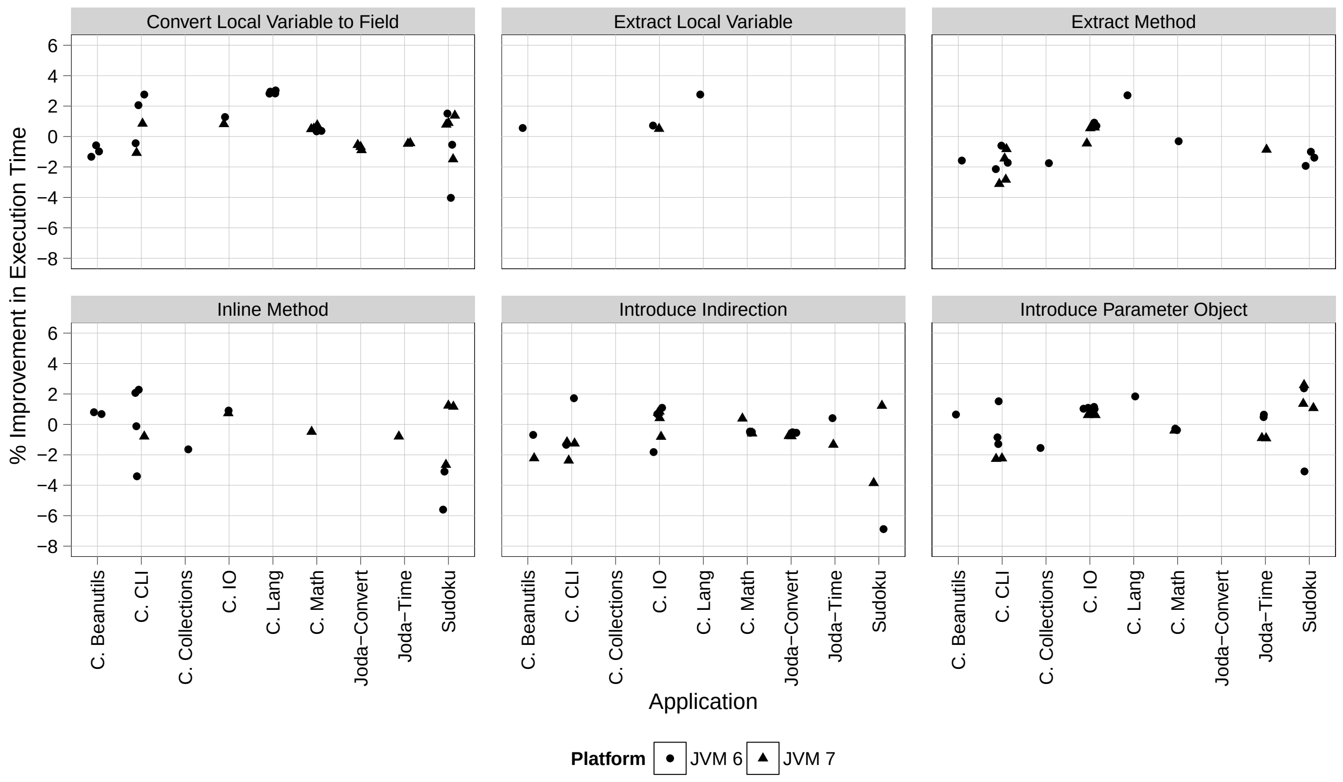

Figure 4: Impacts on execution time of applying refactorings ($\alpha \leq 0.05$).

a refactoring will cause a change in energy usage when run on JVM 6 but not when run on JVM 7, and vice versa.

3.3 RQ3—Predicability

Based on our results from investigating RQ2, which suggest that the context in which a refactoring is applied may determine its energy usage, we decided to investigate whether or not it is possible to predict the impact of applying a refactoring from information that is more easily gathered than energy usage.

3.3.1 Execution Time

One of the most common questions that is asked about energy usage is whether or not it is strongly correlated with execution time. Intuitively, it makes sense that they would be strongly correlated; the longer a program runs, the more energy it consumes. However, this is not necessarily true [11]. It is possible for certain components such as disk drives or Wi-Fi radios to consume significant amounts of energy even during short executions. This is why our LEAP node's ability to profile not only the CPU, but the disk and memory as well, is especially useful. With its capabilities, we can observe the energy costs of the additional components.

We computed a correlation of 0.81 between the execution times and energy usages of our subjects using Kendall's tau, with $\alpha = 0.05$. This indicates that there is a moderately strong positive correlation between execution time and energy usage. Although this result fits with the accepted view, it was surprising for us. Because our applications are CPU-bound and do not use the network or expensive sensors, we expected a much stronger correlation.

To gain some additional insight into whether changes in execution time can be used to predict changes in energy consumption, we identified the cases where applying a refactoring significantly changes execution time. To do this, we used a procedure similar to the one we used to determine when a refactoring significantly impacted energy usage. Again, we used the Mann-Whitney-Wilcoxon test with $\alpha = 0.05$ to identify the cases where execution time was significantly changed. We then computed the percentage difference between the means of execution times of the original and refactored versions in each subject.

Figure 4 presents the results of these calculations. Like for Figure 3, Figure 4 is composed of 6 subfigures, one for each refactoring. The x-axis shows our considered applications and the shape of the points shows whether the subject was executed using JVM 6 or JVM 7. The only difference is that the y-axis now shows the percentage improvement in execution time rather than the percentage improvement in energy usage.

Looking at the results of this comparison, we found that $\approx 11\,\%$ of the time (44 of the 394 cases), a significant change in one measure was not matched by a significant change in the other measure. For the subjects run on JVM 6, there are 76 cases where either energy usage or execution time was significantly impacted by applying a refactoring. For 6 of those cases, there was a significant change in energy usage but not a significant change in execution time; for 16 of those cases, there was a significant change in execution time, but not a significant change in energy usage. In the remaining 54 cases, there was both a significant change in energy usage and in execution time. Similarly, for the subjects run on

JVM 7, there were 16 times when there was a significant change in execution time but not a significant change in energy usage, 6 times when there was a significant change in energy usage but not change in execution time, and 43 times when there was a change in both.

Consequently, we believe that, while energy usage and execution time are roughly correlated, execution time alone is unlikely to be an accurate enough predictor of energy usage. A more complex model is needed to account for the situation that execution time itself is unable to explain.

3.3.2 *Dynamic Execution Counts*

In addition to looking at overall execution times, we also considered whether the dynamic execution count of the locations where the refactorings were applied could predict changes in energy usage. Again, we computed Kendall's tau to check for a correlation. We computed a correlation score of -0.04 with $\alpha = 0.05$. This means that there is essentially no correlation between energy usage and the number of times the location where refactoring is made is executed. As such, we believe that execution counts are a poor predictor of energy usage impacts.

3.4 Threats to Validity

One of the most significant threats to the validity of our results is the possibility of energy usage measurement errors either due to imprecise measurements or failing to control for potential sources of noise. To minimize this threat, we chose to use a measurement tool, the LEAP node, that was designed and validated by an experienced engineering group [20]. Furthermore, we improved the design of the tool to reduce several possible sources of unwanted noise. We also made sure to offload as much work as possible from the LEAP node and to disable all unessential services and programs to minimize potential sources of noise. And finally, we executed our subjects (original and refactored versions) in sequence to ensure that the environments in which they executed were as similar as possible.

Another threat to validity is that we considered 197 instances of only 6 refactorings. Although this is a significantly larger number of subjects than were used in similar studies (e.g., [7, 13, 19]), it may be an insufficient number of data points to allow our conclusions to generalize to all potential contexts in which the refactorings can be applied. Similarly, considering only 2 platforms may prevent our conclusions from generalizing to all environments where Java applications may be executed.

4. RELATED WORK

The energy impacts of several different decisions made by software engineers have been previously investigated in several small empirical studies. Like our study, these studies indicate that the decisions that developers make can have a significant impact on the energy usage of their application.

da Silva et al. measured the performance and energy impacts of inlining methods on 3 embedded Java applications (an address book, a game called Sokoban, and an MP3 audio decoder) [8]. The results of their study agree with our observations; inlining methods can increase energy usage in some instances while decreasing it in others.

Sahin et al. investigated changes in energy consumption due to applying design patterns. They considered a total of 15 patterns, 5 from the each of the standard design pattern categories: creational, structural, and behavioral [19]. Their results indicate that the impacts on energy usage of applying design patterns vary. They observed cases where applying a design pattern increased energy usage, cases where it decreased energy usage, and cases where energy usage was unchanged. In addition, they found that the impacts are also not consistent within a design pattern category. Other studies on design patterns showed similar results [7, 13].

Bunse et al. compared energy usage impacts of choosing among different sorting algorithms executed on an embedded system [6]. They found that the algorithms indeed use different amounts of energy. Additionally, they demonstrated that there is no correlation between time complexity and energy usage of sorting algorithms.

Pathak et al. worked on popular smartphone apps to examine where the energy is spent inside the apps [17]. They implemented and evaluated *eprof*, the first fine-grained energy profiler for smartphone apps on mobile Android and Windows operating systems. One of their outcomes is that two versions of the same app might have significantly different energy consumption behavior.

Arunagiri et al. performed a comparative study to solve the global stereo matching problem in terms of performance and energy consumption [4]. Their result suggests that stereo matching with the graph cut algorithm is a lot better than stereo matching with simulated annealing for both terms they consider.

Using CPU energy usage measurements, Amsel et al. compared the energy consumption impacts of selecting between different software systems that achieve the same purpose [2]. In particular, they measured energy usage of several popular Internet browsers including Internet Explorer, Mozilla Firefox, and Google Chrome. Their results showed that Internet Explorer was most energy efficient.

Manotas et al. analyzed the impacts of choosing different web servers (Mongrel, Puma, Thin, and WEBrick) on the energy consumption of a web application [14]. Their experimental results indicate that a web application's energy consumption depends on the web server used to handle its requests. Furthermore, energy efficiency of the web servers changes based on the executed web application's features.

A necessary requirement of all these studies was the difficult task of measuring or estimating energy consumption accurately. Therefore, significant work has focused on this task at various levels. *Hardware instrumentation-based approaches* [18, 20] measure the actual power usage of a system by using physical instrumentation. However, they can be expensive and it is not easy to use specialized hardware, although they provide accurate measurements. *Simulation-based approaches* [5, 9] also can be accurate, but difficult to use like hardware instrumentation-based approaches. They replicate the actions of a processor at the architecture level and estimate energy consumption of each executed cycle by using a cycle-accurate simulator. In *estimation-based approaches* [1, 10, 11, 16], models of energy-influencing features are built to estimate energy usage. Estimation-based approaches are easier and more applicable than the other approaches, but they are usually less accurate.

5. CONCLUSIONS AND FUTURE WORK

We have presented an empirical study that investigates the impacts of applying refactorings on energy usage. As subjects for the study, we used 197 instances of 6 commonly used refactorings to 9 real Java programs of varying sizes and characteristics. In total, we generated over 350 gigabytes of data from 10,300 executions across two separate platforms. The results of this study demonstrate that:

(1) All of the refactorings we considered can statistically significantly impact the energy usage of an application.
(2) All of the refactorings we considered have the potential to both increase and decrease energy usage, with the exception of Extract Local Variable which we only observed to decrease energy usage.
(3) The impacts of the refactorings do not appear to be consistent across or within applications, or across or within platforms.
(4) More commonly used and easily collectible information such as execution time and dynamic execution counts are unlikely to be able to accurately predict the energy impacts of applying a refactoring.

Overall, these results motivate future work in this area. It is clear that refactorings can impact the energy usage of applications, but additional studies are needed to better understand the underlying reasons for the impacts.

Based on these conclusions, there are several potential areas for future work. First, we plan on enlarging the scope of our study. Although we considered a significant number of subjects, adding additional refactorings and subjects, or simply applying each refactoring more times, would potentially allow us to refute or confirm some of our observations (e.g., whether the energy impacts of refactorings are actually consistent with an application). Second, we plan on investigating whether other types of information can be used to accurately predict the impacts of applying refactorings. From the perspective of a software engineer, the answer to this question has a high utility. In most cases, developers do not have access to the type of custom, hardware-based energy profiling tools that we do. As a result, they have no way of identifying whether applying a refactoring has or will increase or decrease energy usage. Providing them with the ability to make accurate predictions about the impacts of applying refactorings would be very useful.

References

[1] N. Amsel and B. 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.

[2] N. Amsel, Z. Ibrahim, A. Malik, and B. Tomlinson. Toward sustainable software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 976–979, 2011.

[3] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10, 2011.

[4] S. Arunagiri, V. J. Jordan, P. J. Teller, J. C. Deroba, D. R. Shires, S. J. Park, and L. H. Nguyen. Stereo matching: Performance study of two global algorithms. pages 80211Z–80211Z–17, 2011.

[5] D. Brooks, V. Tiwari, and M. 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.

[6] C. Bunse, H. Hopfner, S. Roychoudhury, and E. Mansour. Choosing the 'best' sorting algorithm for optimal energy consumption. In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 199–206, 2009.

[7] S. S. 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.

[8] W. G. P. da Silva, L. Brisolara, U. B. Correa, and L. 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.

[9] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. 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.

[10] S. Hao, D. Li, W. Halfond, and R. Govindan. Estimating Android applications' CPU energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 1–7, 2012.

[11] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 92–101, 2013.

[12] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160, 2011.

[13] A. Litke, K. Zotos, A. Chatzigeorgiou, and G. Stephanides. Energy consumption analysis of design patterns. In *Proceedings of the International Conference on Machine Learning and Software Engineering*, pages 86–90, 2005.

[14] I. Manotas, C. Sahin, J. Clause, L. Pollock, and K. Winbladh. Investigating the impacts of web servers on web application energy usage. In *Proceedings of the Second International Workshop on Green and Sustainable Software*, 2013.

[15] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, pages 287–297, 2009.

[16] 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.

[17] A. Pathak, Y. C. Hu, and M. 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.

[18] C. Sahin, F. Cayci, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Towards power reduction through improved software design. In *Proceedings of the Energytech, 2012 IEEE*, pages 1–6, 2012.

[19] C. Sahin, F. Cayci, I. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 55–61, 2012.

[20] D. Singh, P. A. H. Peterson, P. L. Reiher, and W. J. Kaiser. The Atom LEAP platform for energy-efficient embedded computing: Architecture, operation, and system implementation. 2010.

[21] A. Vargha and H. D. Delaney. A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[22] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported—an Eclipse case study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, 2006.