SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework

Irene Manotas
Computer Science
University of Delaware
Newark, DE 19716, USA
imanotas@udel.edu

Lori Pollock
Computer Science
University of Delaware
Newark, DE 19716, USA
pollock@udel.edu

James Clause
Computer Science
University of Delaware
Newark, DE 19716, USA
clause@udel.edu

ABSTRACT

Reducing the energy usage of software is becoming more important in many environments, in particular, battery-powered mobile devices, embedded systems and data centers. Recent empirical studies indicate that software engineers can support the goal of reducing energy usage by making design and implementation decisions in ways that take into consideration how such decisions impact the energy usage of an application. However, the large number of possible choices and the lack of feedback and information available to software engineers necessitates some form of automated decision-making support.

This paper describes the first known automated support for systematically optimizing the energy usage of applications by making code-level changes. It is effective at reducing energy usage while freeing developers from needing to deal with the low-level, tedious tasks of applying changes and monitoring the resulting impacts to the energy usage of their application. We present a general framework, SEEDS, as well as an instantiation of the framework that automatically optimizes Java applications by selecting the most energy-efficient library implementations for Java's Collections API. Our empirical evaluation of the framework and instantiation show that it is possible to improve the energy usage of an application in a fully automated manner for a reasonable cost.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.2.6 [**Programming Environments**]: Integrated environments

General Terms

Theory, Measurement

Keywords

Energy usage, software optimization, analysis framework

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 power consumption can be reduced through designing computer architectures that are more energy efficient (e.g., [11, 12, 24, 26, 40, 47]), developing compiler optimizations targeting energy usage (e.g., [19–22, 25, 27, 41, 43]), improving operating systems to help manage energy usage (e.g., [14, 36–38, 52]), and designing hardware and batteries with power consumption in mind (e.g., [1, 7, 13]).

We believe that similar to other optimization targets such as execution time or memory usage, not all improvements can be achieved automatically by lower-level systems and hardware. Unfortunately, few software developers design and implement applications with consideration for their energy usage. Our recent interviews with 18 professional software developers revealed that this is due to two primary reasons: developers (1) do not understand how the software engineering decisions they make affect the energy consumption of their applications, and (2) lack the tool support to help them make decisions or change their code to improve its energy usage.

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 design patterns [8, 29, 42], method inlining [10], choosing web servers [30], selecting among web browsers [3], and using various implementations of an algorithm [6] can each impact energy usage. The results of these studies support our belief that software engineers can indeed help reduce energy consumption by considering the energy impacts of the decisions they make on a daily basis.

Although the knowledge gained from empirical studies can increase our understanding of potential energy-related "bugs", we believe that simply providing 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). A recent survey on Android fragmentation showed that there are over 10,000 possible combinations of hardware and software that can run Android applications [34]. 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.

This paper describes the Software Engineer's Energy-optimization Decision Support framework (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.

In this paper, we present how SEEDS can be instantiated by describing how we used it to create the SEEDS API Implementation Selector (SEEDS$_{\text{api}}$), a tool that optimizes Java applications that use the Java Collections Framework (JCF). SEEDS$_{\text{api}}$ automatically selects the most energy efficient implementations of the Collections application programming interface (API) to use at each location where a collection object is allocated. In short, SEEDS$_{\text{api}}$ automatically (1) generates application versions implementing many different alternative combinations of API implementation choices for all the object instantiation locations in the application, (2) performs power-monitored executions for a given test suite on all the generated versions, (3) analyzes the collected energy usage data to identify the best combination of API implementation choices per object allocation location, and (4) generates an optimized version of the application based on API implementation decision-making.

An evaluation comparing the energy usage of 7 unmodified applications and the corresponding optimized versions created by SEEDS$_{\text{api}}$ reveals that the framework is able to effectively improve the energy usage of applications without requiring the software engineer to provide more than the application, API implementations, and test suite.

The main contributions described in this paper are:

- 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.
- 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.
- An evaluation of the effectiveness and cost of SEEDS through the use of SEEDS$_{\text{api}}$ on a set of open source projects.
- 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.

The remainder of this paper is organized as follows: Section 2 describes SEEDS, our framework for optimizing energy usage. Section 3 presents SEEDS$_{\text{api}}$, our instantiation of SEEDS. Section 4 presents our empirical evaluation of SEEDS and SEEDS$_{\text{api}}$ including our methodology, data, and analysis. Finally, Sections 5 and 6 discuss related work and present our conclusions and future work.

2. THE SEEDS FRAMEWORK

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.

(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.

Figure 1 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.

2.1 Inputs

As the figure shows, 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. 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, whether to cache the result of a computation, etc. 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 reused and frees developers from the task of recomputing them for each new application that they want to improve. The method for transforming abstract potential changes into concrete changes for the given application is described in Section 2.2.

The **optimization parameters** are constraints on where SEEDS should consider making potential changes. For example, a developer could restrict the application of a refactoring to only a certain subset of the application or only allow switching algorithms if the algorithm's inputs are larger than a given threshold. In addition, the optimization parameters can also include guidance about how changes should be applied. This allows software engineers to encode their domain-specific knowledge and intuition into the framework. 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 applying various refactorings based on recommendations from a tool or documentation that indicates that applying that refactoring improves code readability and maintainability.

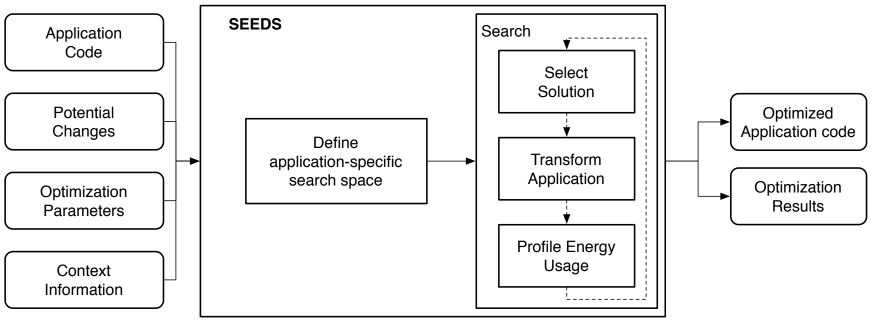

Figure 1: **Overview of SEEDS.**

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 2.3) 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.

2.2 Creating Application-Specific Search Space

As we mentioned previously, the changes in the set of potential changes are abstract rather than concrete. SEEDS' first task then 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.

To calculate 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 optimization parameters.

2.3 Search: Select, Transform, Profile

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.

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. For example, the selection strategy could be to select a solution in a random manner, based on a heuristic, using a genetic algorithm, etc. In Section 3.2, we describe the selection strategy implemented for an instantiation of the SEEDS framework. 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 is a fifth input to the framework.

The second step of the search task is to transform the application by applying the chosen solution. 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, including hardware-based approaches using physical instrumentation and monitoring (e.g., [42, 46, 50]), simulation-based approaches that replicate the actions of a processor and estimate energy consumption of each executed cycle by using a cycle-accurate simulator (e.g., [5, 16, 31]), or estimation-based approaches that model energy-influencing features to estimate energy usage (e.g., [2, 17, 33]).

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.

2.4 Output

The output of SEEDS includes the optimized application code and the optimization results. 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 different given parameters). When given a specific combination of inputs, the application may not be improved in terms of energy usage, if then SEEDS returns the original application version as output . In our evaluation, this situation occurred once. 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. The optimization results presented to the developer include by how much the energy consumption of the application is decreased when comparing the optimized version versus the original application.

3. INSTANTIATING SEEDS

To evaluate SEEDS, it is necessary to create an instantiation. As we mentioned in Section 2, there are many common decisions that software engineers make that could be the target of an instantiation of SEEDS. We created an implementation of SEEDS called SEEDS$_{\text{api}}$ 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, 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 target the choice of the Collections API implementation 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 are the first to look at the choice of API implementations, we conducted an initial feasibility study to determine whether the choice of API implementations does in fact impact the energy usage of an application, before actually implementing SEEDS$_{\text{api}}$.

The remainder of this section discusses our preliminary study and how we instantiated SEEDS to create SEEDS$_{\text{api}}$.

3.1 Preliminary Study

The goal of our preliminary study was to determine if changing implementations of the Collections API can impact the energy usage of an application.

To answer this question, we created 13 versions of a publicly available micro-benchmark.[1] 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).

Each of the 13 versions of the benchmark we created uses a different concrete implementation of the `Collection` interface. The first column of Table 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. (See Section 3.2 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 1, the second and fourth columns, *# Better* and *# Worse* show, given the current implementation in the first column, the number of times switching to an another implementation improves energy usage ($\alpha = 0.05$) and the number of times switching to another concrete implementation wors-

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

Table 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

ens 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 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.

3.2 SEEDS$_{\text{api}}$

Based on the results from our preliminary study on the impact of switching implementations of the Collections API, we went forward with creating SEEDS$_{\text{api}}$. A high-level overview of SEEDS$_{\text{api}}$ is shown in Figure 2, 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 Section 2.

Application code. SEEDS$_{\text{api}}$ is designed to optimize Java applications. Therefore, it accepts as input Java applications that use the Collections API.

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`. Note that SEEDS$_{\text{api}}$ can consider changes between any implementations that implement the same Collections API. Currently, the tool includes all implementations from the JCF as well as all implementations of `Collection` from Javolution,[2] fastutil,[3] Apache Commons Collections,[4] Goldman-Sachs Collections,[5] and Google's Guava libraries.[6]

We have also built an automated tool that automatically extracts potential changes from the set of libraries. 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.

Context information. To use SEEDS$_{\text{api}}$, developers must provide a test suite as part of the context information. 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.

Define application-specific search space. 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

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

[3]`http://fastutil.di.unimi.it`

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

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

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

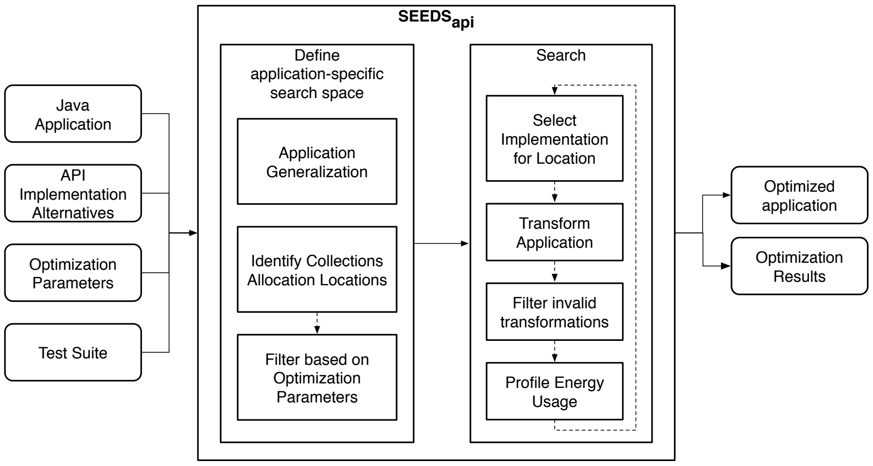
SEEDS$_{api}$
Java Application
API Implementation Alternatives
Optimization Parameters
Test Suite
Define application-specific search space
Application Generalization
Identify Collections Allocation Locations
Filter based on Optimization Parameters
Search
Select Implementation for Location
Transform Application
Filter invalid transformations
Profile Energy Usage
Optimized application
Optimization Results

Figure 2: **Overview of SEEDS$_{api}$.**

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$_{api}$ 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`.

After generalization, SEEDS$_{api}$ analyzes the application to identify the locations where instances of the Collections API are created. 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, SEEDS$_{api}$ uses the T.J. Watson Libraries for Analysis (WALA) [48] to implement a def-use analysis that tracks backwards from the target object of the `invokespecial` to the `new` where it was created.

After identifying the locations in an application where collection objects are allocated, SEEDS$_{api}$, determines how many potential changes could be applied at each location. For example, if an instance of the Set interface is being created, SEEDS$_{api}$ 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, constitues SEEDS$_{api}$'s application-specific search space.

Select solution. 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 implement one of the simplest search strategies possible: an exhaustive exploration of all possible applications on a single concrete change.

Our search strategy starts by identifying the most energy efficient implementation choice at each location in the program where an object that implements the Collections API is created. SEEDS$_{api}$ identifies the most efficient implementation choice at each location by applying each concrete change 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. Note that we are considering each location separately. After identifying the most efficient implementation at each object allocation location, SEEDS$_{api}$ creates one additional version where the most efficient change at each location is applied. Finally, the selection strategy used by SEEDS$_{api}$ 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 version that results in the largest decrease in energy usage. If none of the changed versions is a statistical improvement over the original, unmodified application, the original version is returned instead. In this way, SEEDS$_{api}$ is guaranteed to never produce an optimized version that performs worse than the original application.

Although this strategy is simple, it results in optimized applications that are more energy efficient than the original applications. In our experiments, SEEDS$_{api}$ improved the energy usage of 6 of our 7 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.

Transform application. To apply the selected changes, SEEDS$_{api}$ uses ObjectWeb's ASM bytecode rewriting library[7] to change the types of `new` and `invokespecial` bytecodes that correspond to the locations of the selected changes. We chose to directly modify the application's bytecode, rather than its source code, so that SEEDS$_{api}$ 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 transfor-

[7]`http://asm.ow2.org`

Table 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

mation has not broken the functionality of the application.

Profile energy usage. To profile the amount of energy consumed when executing an application, we used a Low Power Energy Aware Processing (LEAP) node [46]. Our LEAP node is 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 data acquisition (DAQ) card (National Instruments USB-6215) that samples the amount of power consumed by the component at a rate of 10 kHz ($\approx$ 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.

4. EMPIRICAL EVALUATION

Our evaluation of SEEDS focuses on evaluating the effectiveness of using an instantiation, namely 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?

4.1 Experimental Subjects

The primary goal of 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 7 Java applications that use the Collections API. We also selected these programs because they have been used by many researchers and they are representative of applications that use the JCF. In addition, because SEEDS$_{\text{api}}$ requires a test suite, we needed to select applications that have an associated test suite.

Table 2 describes the seven 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 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 the three different public repositories: (1) Software-artifact Infrastructure Repository (SIR),[8] which provides a variety of open-source projects for empirical software engineering, (2) SourceForge,[9] a popular repository for open-source projects, and (3) Apache Commons,[10] a collection of reusable components.

4.2 RQ1: Effectiveness

In our preliminary study (see Section 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 is to determine whether we can achieve the same type of improvements in real applications in a fully automatic manner.

To answer this question, we created 2 optimized versions of each of our subjects using SEEDS$_{\text{api}}$, one where SEEDS$_{\text{api}}$ was allowed to use only Collections implementations from the JCF and one where SEEDS$_{\text{api}}$ was allowed to use Collections implementations from all of its included libraries. For the cases where SEEDS$_{\text{api}}$ was able to optimize the applications (i.e., it returned a version different than the original), we ran the original and optimized versions on the LEAP node 10 times. Then we used the Kruskal-Wallis test to determine whether there is a statistically significant difference in the amount of energy usage 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

[8]http://sir.unl.edu
[9]https://sourceforge.net
[10]http://commons.apache.org

Table 3: **SEEDS$_{\text{api}}$ effectiveness in improving energy usage.**

	% Improvement	
Application	JCF Only	ALL
Barbecue	17^*	17^*
Jdepend	3^*	6^*
Apache-xml-security	$5^\dagger$	$5^\dagger$
Joda-Time	8^*	$9^\dagger$
Commons Lang	$10^\dagger$	$13^\dagger$
Commons Beanutils	—	—
Commons CLI	2^*	2^*

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

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 optimized versions to determine how effective SEEDS$_{\text{api}}$ was at improving the energy usage of the applications.

Table 3 shows the data we generated to investigate the effectiveness of 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 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 3.2), *ALL*. Note that a dash (—) indicates that SEEDS$_{\text{api}}$ was unable to optimize the application. A $*$ indicates that the optimized version was constructed using only one concrete change, and a $\dagger$ indicates that the optimized version was constructed by applying the best individual change at each location.

There are several interesting observations that we can make from this data. First, SEEDS$_{\text{api}}$ was effective at automatically improving the energy usage of our subjects. For all but one application, 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 search strategy that only considered changes applied in isolation.

Second, the optimized versions produced by SEEDS$_{\text{api}}$ include versions that contain only one change (7 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 location. The fact that approximately 60 % of the time, the most efficient version contains only a single change, suggests that there are 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.

4.3 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 SEEDS$_{\text{api}}$ to explore the questions:

RQ2a. How does the effectiveness of the energy optimization change with more choices?
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 implementation the most energy efficient?

RQ2a: We can use our results to also answer the question "How does the effectiveness of the energy optimization change with more choices?" As Table 3 shows, the effectiveness of SEEDS$_{\text{api}}$ only slightly increases when considering all possible implementations of the Collections API rather than just the implementations from the JCF. For 4 subjects, adding the additional implementations had no impact on the performance of the SEEDS$_{\text{api}}$. For the remaining 3 subjects, energy usage was improved, but the magnitude of the improvement was 3 % 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 [2]. 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 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 123 total locations where an instance of the Collections API is allocated. When only implementations from the JCF are considered, 56 % of the time (69 cases), switching away from the original implementation resulted in a decrease in energy usage. Similarly, when all possible implementations were considered, 72 % of the time (89 cases) switching away from the original implementation improved energy usage. These results motivate the need for SEEDS as they show that developers are only infrequently 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, SEEDS$_{\text{api}}$ chooses among 157 distinct implementations of the Collections API. Figure 3 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, `ArrayList`, `Vector`, and `HashSet` (all from the JCF are the implementations that were most frequently the most efficient.

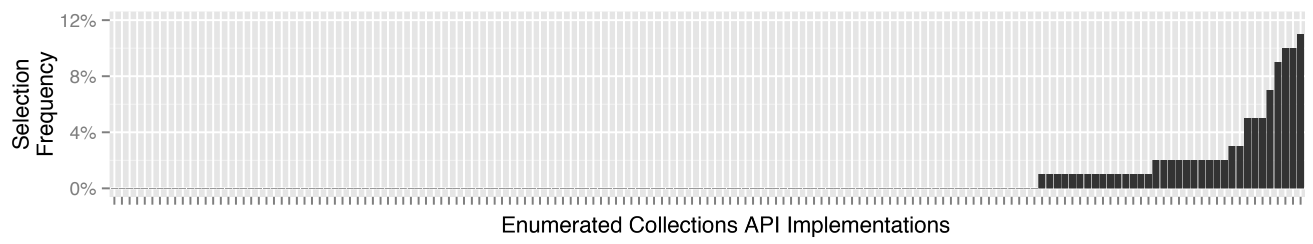

Figure 3: **Percentage of time each Collection API implementation was selected as the most energy efficient.**

As Figure 3 shows, there is not a single implementation that is always the most energy efficient. Moreover, it shows that there are 35 implementations 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 35 implementations that were the best at least once, there were far more implementations that were never the most energy efficient. This information could be used to help direct a search strategy.

4.4 RQ3: Cost

The question "Can SEEDS provide decision-making support to the software engineer with regard to energy consumption implications at a reasonable cost?" was addressed by recording the times to perform each step of the framework for each of the 7 subject applications.

Table 4 presents the estimated costs, in terms of wall clock time, for optimizing applications. In the table, the first column, *Application*, shows the name of each application. The second column, *Exe.*, shows the amount of time 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 SEEDS$_{\text{api}}$), |*Search*|. The fifth column, *Analysis*, shows the time 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*, and *Analysis* and *Cost*, respectively when considering all implementations included in SEEDS$_{\text{api}}$ (*ALL*). As shown in the table, the total cost ranges from 3 h to 64 h for JCF Only and from 4 h to 175 h hours for ALL.

By far the largest portion of the cost of using 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 SEEDS$_{\text{api}}$ can easily be tailored to fit a software engineer's specific circumstances. As Table 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 expores 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 an exhaustive search strategy and ran the entire test suite. Instead, we could have used a non-exhaustive strategy and only executed part of test suite in order to reduce the cost of using SEEDS$_{\text{api}}$.

4.5 Threats to 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 many possible search strategies that may provide better energy usage; and although our strategy is simple, it does indeed show that SEEDS can result in optimized applications that are more energy efficient than the original applications. 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.

For our evaluation, we selected 7 Java applications, used their associated test suites, and chose 6 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 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.

Table 4: **Cost to automatically optimize an application.**

Application	Exe. (s)	# Reps.	JCF Only |Search|	JCF Only Analysis (hrs)	JCF Only Cost (hrs)	ALL |Search|	ALL Analysis (hrs)	ALL 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

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.

5. RELATED WORK

The most closely related work is the one presented in [15], 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 [51] and high performance computing [39, 49] as well as for general purpose languages and platforms (e.g., [45]).

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 [45]. 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 is a group of work that has attempted to identify the underlying causes of energy consumption by empirically investigating the impact of various software development decisions. More specifically, researchers have investigated the impacts of refactorings [10], design patterns [8, 29, 42], sorting algorithms [6], web servers [30], programming models [9, 44], and lock-free data structures [23] within a single application, in addition to investigating trends in an application's energy consumption among versions [18] and among separate implementations of the same specification [4, 35]. Insights gained from these and similar studies could be integrated into the search component of SEEDS to help it identify more energy-efficient changes more quickly.

Third, there is a significant amount of work focused on accurately measuring energy consumption. Work in this area has been conducted at various levels. *Hardware instrumentation-based approaches* (e.g., [46, 50]) use physical instrumentation (e.g., soldering wires to power leads) to measure the actual power usage of a system. *Simulation-based approaches* (e.g., [16, 31, 32]) use a cycle-accurate simulator to replicate the actions of a processor at the architecture level and estimate energy consumption of each executed cycle. Finally, *estimation-based approaches* (e.g., [2, 5, 17, 44]) build models of energy-influencing features and use such models to estimate energy usage.

Finally, researchers have begun to build on the accurate measurement work mentioned above to provide source code-level feedback on energy consumption to developers [28]. 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.

6. CONCLUSIONS AND FUTURE WORK

SEEDS is the first known framework for helping software engineering make decisions with regard to energy usage of their application. Our empirical results show that using such automation can indeed improve energy usage of real applications without requiring the software engineer to deal with low-level energy profiling tools and analyses. Instantiating SEEDS to make decisions about which library to choose showed up to 17% energy usage improvement. While the tedious work is hidden from the developer, the collecting and processing of power samples can take many hours depending on the test suite execution time, number of repetitions, and the search space. However, optimizing the energy usage will likely be done infrequently and these costs can be tailored to the software engineer's specific circumstances.

We plan to investigate more advanced search strategies and other instantiations of SEEDS. Specifically, we will investigate using SEEDS to make other kinds of software engineering decisions.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

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

[2] 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.

[3] 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.

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

[8] 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.

[9] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 831–850, 2012.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S. Gotz, C. Wilke, M. Schmidt, S. Cech, and U. Assmann. Towards energy auto-tunning, 2010.

[16] 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.

[17] 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.

[18] A. 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] N. Hunt, P. Sandhu, and L. Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures*, pages 63–70, 2011.

[24] 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.

[25] 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.

[26] 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.

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

[28] D. Li, S. Hao, W. G. J. Halfond, and R. 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.

[29] 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.

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

[31] T. Mudge, T. Austin, and D. Grunwald. The reference manual for the Sim-Panalyzer version 2.0. `http://web.eecs.umich.edu/~panalyzer/`.

[32] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha. Hybrid simulation for embedded software energy estimation. In *Proceedings of the 42nd annual Design Automation Conference*, pages 23–26, 2005.

[33] 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.

[34] OpenSignal. Android fragmentation visualized. `http://opensignal.com/reports/fragmentation-2013/`, 2013.

[35] 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.

[36] 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.

[37] 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.

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

[39] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. 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.

[40] 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.

[41] A. Rangasamy, R. Nagpal, and Y. 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.

[42] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. E. Kiamilev, L. 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.

[43] 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.

[44] C. Seo, S. Malek, and N. 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.

[45] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–418, 2009.

[46] 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. `http://lasr.cs.ucla.edu/deep/FrontPage?action=AttachFile&do=view&target=leapwhitepaper.pdf`, 2010.

[47] 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.

[48] T. J. Watson Libraries for Analysis (WALA). URL `http://wala.sf.net`. Accessed: September 10, 2013.

[49] A. Tiwari, J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame. Auto-tuning full applications: A case study. *International Journal of High Performance Computing Applications*, 25(3): 286–294, 2011.

[50] watts up. `https://www.wattsupmeters.com/secure/index.php`.

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

[52] 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.