Energy-Directed Test Suite Optimization

Ding Li,* Cagri Sahin,† James Clause,† and William G.J. Halfond*
*Computer Science Department
University of Southern California, Los Angeles, CA, USA
†Computer and Information Sciences Department
University of Delaware, Newark, DE, USA
ding@usc.edu, cagri@udel.edu, clause@udel.edu, halfond@usc.edu

***Abstract*—Post-deployment, in situ testing and validation techniques have become an important means of ensuring the reliability of mobile and embedded systems. However, these techniques do not take into consideration the amount of energy they consume, which is an issue of paramount concern for systems with limited energy budgets. In this paper we propose a new test suite minimization approach that allows developers to generate energy-efficient, minimized test suites. The approach is based on encoding minimization problems as integer linear programming problems. Our empirical evaluation shows that, compared to traditionally generated minimized test suites, the test suites generated by our approach are equally effective in terms of their test coverage, but can realize energy savings of up to 90 %.**

***Index Terms*—Energy use, Testing, Test suite optimization, Post-deployment validation**

I. Introduction

Ensuring reliable software operation in mobile and embedded systems is extremely challenging. Common characteristics of such systems, including severe resource constraints and operation in diverse and unpredictable environments, often result in uncommon and unexpected failures that manifest only in the field (e.g., [1]–[4]). Under these circumstances, pre-deployment validation efforts are insufficient—they do not accurately capture or represent the behavior or environment where the software will actually execute. Consequently, the use of post-deployment, in-field testing and validation has become mandatory for ensuring the correct operation of many systems.

Because of the need to perform these post-deployment validation activities, researchers in the software engineering community have developed techniques for performing testing, analysis, and debugging in situ (e.g., [5]–[7]). Although these techniques are useful in many scenarios, they do not consider energy consumption, which is a key area of concern for mobile and embedded systems. As a result, their use can rapidly drain the battery power of these systems, compromising the systems' functionality by leaving them with limited power [8]. Given this potential outcome, it is necessary for developers to strike a balance between fulfilling all of a system's in situ validation requirements and preserving its limited battery power.

Unfortunately, developers lack the information needed to identify the appropriate balance between testing thoroughness and energy consumption. For example, a developer will often try to minimize a test suite that is to be deployed and run in situ by selecting as few test cases as possible while maintaining the structural coverage level of the original test suite. Current techniques provide this ability (e.g., [9], [10]), but do not allow the developer to also minimize the test suites for energy consumption. As we show in Section IV, the difference between the energy consumption of minimized test suites with the same coverage can be quite large—in many cases over 80 %! The fact that existing techniques cannot leverage this difference, or even provide developers with test selection guidance, can lead to sub-optimal usage of a system's limited battery power.

In this paper we present our ongoing research into the impacts of energy-awareness on post-deployment verification techniques. We define a new test suite minimization technique that supports traditional minimization goals [11], such as maintaining fault finding capability and minimizing execution time, but also prioritizes reducing the amount of energy consumed by a test suite. We implemented this technique in a prototype tool, the Energy-directed Test Suite Optimizer (EDTSO), and used it to perform an empirical study of the usefulness and effectiveness of energy-directed test suite minimization. In the study, we carried out a series of experiments designed to answer several research questions about the technique's feasibility and effectiveness.

Overall, the results of our evaluation are very positive. They demonstrate (1) the need for our approach, and (2) its effectiveness at producing energy-efficient, minimized test suites. More specifically, they show that, in some cases, our approach produces minimized test suites that consume less than 90 % of the energy consumed by equivalent test suites generated by traditional minimization approaches. We believe that, with further research and development, EDTSO has the potential to become a vital tool for mobile and embedded systems developers, as it will allow them to perform post-deployment validation without comprising the functionality of their applications.

This work makes the following contributions:

- The definition of a new approach for optimizing the energy usage of test suites used for post-deployment validation.
- EDTSO, a prototype implementation of the approach that optimizes test suites for Android applications.
- An empirical evaluation of the approach that demonstrates its feasibility and effectiveness.

The remainder of this paper is organized as follows: Section II presents necessary background about test suite minimization and a motivating example that is used throughout the paper. Section III describes the details of our approach.

978-1-4673-6267-2/13 © 2013 IEEE

Section IV presents our prototype implementation and our empirical evaluation. Finally, Sections V and VI discuss related work and present our conclusions and future work.

II. Background and Motivating Example

In this section, we provide background information on test suite minimization. We then motivate the need for energy directed test suite optimization by demonstrating, with an example, how traditional approaches for test suite minimization can create minimized test suites that are not optimal with respect to their energy usage.

To help validate an application, developers will typically use a test suite (i.e., a collection of tests that encode the expected behavior of the application for a set of inputs). There are many situations where developers would like to reduce the number of test cases in a test suite (e.g., because the entire test suite take too long to execute, or certain tests exercise unchanged portions of the application) without compromising the effectiveness of the test suite. Approaches to solve this problem are referred to as test suite minimization techniques.

In general, the goal of test suite minimization is to reduce the size of a test suite while maintaining its effectiveness. Harrold and colleagues formalized this intuitive understanding as follows [12]:

Definition 1 (Test suite minimization): Given:

(1) a program P
(2) a test suite T for P
(3) a set of test requirements $\{r_1, \ldots, r_n\}$, derived from a minimization criteria, that must be satisfied to "adequately" test P
(4) subsets of T, $T_1, \ldots, T_n$ such that $\forall\, t \in T_i \,|\, t$ satisfies r_i

find a representative set, T', comprised of test cases from T, that satisfies all r_i.

Under this definition, T' is equivalent to a hitting set (vertex cover) for all T_i and a minimal T' is equivalent to the minimal hitting set.

Coverage metrics (e.g., structural coverage, feature coverage, etc.) are often used as a mechanism for assessing the adequacy of a test suite. As a result, a common usage of test suite minimization is to reduce the number of test cases in a test suite while maintaining the original level of coverage. To carry out coverage-based minimization, a developer would first collect coverage information for each test case in a test suite. For example, the statements executed by each test case. Then the developer would use a minimization technique to find a subset of the test cases in the test suite that achieve the same coverage as the complete test suite. Generally, this is done by examining the coverage contribution of each test case and eliminating test cases that are redundant or do not add any additional coverage as compared to previously selected test cases.

To illustrate the test minimization process, consider the Android application excerpt shown in Table I. This code obtains the user's current location (Line 2) and checks to see if the user has moved more than a certain distance from their prior location (Line 4). If so, the application launches a new search based on the current location (Line 5). The application also checks to see if the display is showing the last line of the available data and retrieves more downloaded data to populate the display (Line 8). Finally, if the screen is active, the display is refreshed with the new data (Line 11).

Table I also shows coverage information for the application's test suite. The columns under the heading *Test cases* show, for each test case in the test suite, whether it covers each statement in the application. A check mark (✓) indicates that a statement is covered by a test case while a blank space indicates that a statement is not covered. For example, Line 5 is covered by test cases t_2, t_4, and t_5, but not covered by t_1 and t_3. Taken together, t_1 through t_5 achieve $100\,\%$ statement coverage for the application (i.e., they execute every statement).

There are several possibilities for creating a minimized test suite that maintains the same level of coverage as the complete test suite. In our example, we consider two such test suites. The first is to choose test cases t_1, t_2, and t_3; we will refer to this possibility as T_1. The second is to chose test cases t_4 and t_5; we will refer to this possibility as T_2. Both T_1 and T_2 achieve the same coverage as the complete test suite, but contain fewer test cases than the original test suite.

In practice, T_2 is more likely to be produced by a minimization tool as it contains fewer test cases than T_1. However, in this example, choosing T_2 instead of T_1 leads to much high energy consumption. To illustrate why this is the case, consider the energy costs associated with each test case, which are shown in the final row in Table I. These numbers show the amount of energy, in milli-Joules (mJ), consumed by each test case when it executes on a specific hardware platform.

By summing the energy cost of each test case in T_1 and T_2, we can calculate the test suites' total energy cost of $1.4\,$mJ and $2.8\,$mJ, respectively. As can be seen, even though T_2 contains fewer test cases than T_1, it consumes twice as much energy. This difference can be attributed to the fact that in T_2 both test cases power up the 3G radio to conduct a search (line 5)—an operation that requires a large amount of energy—while T_1 only includes one test case (t_2) that powers up the 3G radio.

Our energy-directed test suite minimization approach (Section III), addresses this problem. It incorporates energy consumption data into the minimization process to provide developers with the ability to create test suites that consume as little energy as possible.

III. Our Approach

The goal of our test suite minimization approach is to enable developers to quickly and easily produce energy-efficient, minimized test suites. To accomplish this goal, we have defined a new technique for test minimization that treats energy minimization criteria as first-class citizens of the minimization process. This technique enable developers to optimize test suites with respect to energy-efficiency as easily as they can optimize with respect to more traditional criteria, such as minimizing execution time and maximizing fault detection capability.

Our approach is based on the insight that test suite minimization can be formulated as an integer linear programming (ILP)

Table I
APPLICATION AND TEST CASE DATA FOR OUR MOTIVATING EXAMPLE.

	Test cases				
Coverage	t_1	t_2	t_3	t_4	t_5
`1. public void updateData(Location prior) {`	✓	✓	✓	✓	✓
`2.   Location current = LocationManager.getLocation();`	✓	✓	✓	✓	✓
`3.   SearchData data = dataDisplay.getData();`	✓	✓	✓	✓	✓
`4.   if (prior.distanceTo(current) > MAX_DISTANCE) {`	✓	✓	✓	✓	✓
`5.     data = getNewSearchResults(current);`		✓		✓	✓
`6.   }`	✓	✓	✓	✓	✓
`7.   if (dataDisplay.getRow() == MAX_ROW) {`	✓	✓	✓	✓	✓
`8.     data = getSearchResults(prior, dataDisplay.getRow())`			✓		✓
`9.   }`	✓	✓	✓	✓	✓
`10.   if (screenActive()) {`	✓	✓	✓	✓	✓
`11.     refreshDisplay(data);`	✓			✓	
`12.   }`	✓	✓	✓	✓	✓
`13. }`	✓	✓	✓	✓	✓
Energy consumption (mJ)	0.5	0.7	0.2	1.3	1.5

problem (e.g., [9], [10]). ILP is a type of linear programming (LP), a commonly used method for determining the best possible outcome for a given mathematical model under a set of requirements. Note that while ILP is NP-Hard, in practice, the majority of test suite minimization-based ILP problems are solvable in a reasonable amount of time (see Section IV-F). The primary benefit of representing minimization problems as ILP problems is that the approach is able to produce optimal minimized test suites; as long as the ILP problem can be solved, the approach is guaranteed to produce a test suite that satisfies all of the minimization criteria.

At a high-level, our approach has 4 main steps: (1) gather energy and test coverage data, (2) encode the minimization problem as an ILP problem, (3) pass the encoded problem to a backend ILP solver, and (4) construct a minimized test suite based on the solution provided by the ILP solver.

Like all other minimization approches, our approach requires testers to provide information about each test case in T (i.e., coverage and energy data). Coverage data can be computed by instrumenting the target application, running each test case, and recording the structural coverage (e.g., statement or branch). Energy data can be directly measured [13], estimated using program analysis based techniques [14], or simulated via a cycle-accurate simulator [15]. Both coverage and energy data would be computed for the application before its deployment.

Our mechanism for encoding minimization problems as ILP problems is based on three insights [10]. The first is that a minimized test suite T' for a test suite T can be represented as an array of binary values $A = \left[b_1, b_2, \ldots, b_{|T|}\right]$. A value of true (resp., false) for b_i indicates that the ith test case in T should (resp., should not) be included in T'. The second is that an objective function that minimizes the number of true values in A also minimizes the size of T'. And finally, minimization criteria can be expressed as follows: absolute criteria (i.e., criteria that the minimized test suite must satisfy) can be encoded as linear relationships among the elements of A; relative criteria (i.e., criteria that the minimized test suite should satisfy) can be encoded as part of the objective function mentioned previously.

```
bin: b1, b2, b3, b4, b5;

min: 0.5 b1 + 0.7 b2 + 0.2 b3 + 1.3 b4 + 1.5 b5;

 s1: b1 + b2 + b3 + b4 + b5 >= 1;
 s2: b1 + b2 + b3 + b4 + b5 >= 1;
 s3: b1 + b2 + b3 + b4 + b5 >= 1;
 s4: b1 + b2 + b3 + b4 + b5 >= 1;
 s5:      b2 +      b4 + b5 >= 1;
 s6: b1 + b2 + b3 + b4 + b5 >= 1;
 s7: b1 + b2 + b3 + b4 + b5 >= 1;
 s8:           b3 +      b5 >= 1;
 s9: b1 + b2 + b3 + b4 + b5 >= 1;
s10: b1 + b2 + b3 + b4 + b5 >= 1;
s11: b1 +           b4      >= 1;
s12: b1 + b2 + b3 + b4 + b5 >= 1;
s13: b1 + b2 + b3 + b4 + b5 >= 1;
```

Figure 1. ILP encoding of our motivating example.

As a specific example of how a minimization problem can be encoded as an ILP problem, consider again Table I and assume that a developer wants to create a minimized test suite that maintains the same coverage as the original but that minimizes energy usage. In this minimization problem, there is one absolute criteria (i.e., that the minimized test suite must have the same coverage as the original) and one relative criteria (that the minimized test suite should be as energy efficient as possible).

Fig. 1 illustrates how this minimization problem would be encoded by our approach as an ILP problem using the LP file format.[1] We chose to use the LP file format because it is understood by many ILP solvers. This allows us to easily switch between ILP solvers if necessary.

[1] http://lpsolve.sourceforge.net/5.5/lp-format.htm

In the figure, Line 1 shows how the test cases are represented as an array of binary values. The original test suite for this example contains 5 test cases so there are correspondingly 5 Boolean variables ($b_1, \ldots, b_5$). Line 2 shows how the objective function is used to express the desire to create a minimized test suite that is as small as possible (i.e., by minimizing the sum of the Boolean variables). Line 2 also demonstrates how a relative criteria can be expressed by weighting each binary variable in the objective function with energy data. In this example, the relative criteria is that the minimized test suite should consume as little energy as possible. Weighting the binary variables with their corresponding energy costs results in a solution that achieves the best balance between small test suite size and energy efficiency. Finally, Lines 3–15 demonstrate how an absolute criteria can be encoded. In this case the absolute criteria is that the minimized test suite maintains the same coverage as the original (i.e., each statement covered by the original test suite must be covered by the minimized test suite). More specifically, the original test suite in our example covered 15 statements. The criteria that the minimized test suite covers the same statements is expressed as a set of linear relationships, one for each statement that must be covered. For example, Line 13 expresses the fact that, for Statement 11 to be covered, the minimized test suite must include either test case t_1 or t_4. Similarly, Line 15 indicates that Statement 13 can be covered by any of the test cases.

After a minimization problem is encoded, it is passed to an ILP solver. Our choice to use the LP file format means that, in practice, we have a wide range of ILP solvers to choose from; a benefit that can be used to improve the execution time of the approach by running multiple solvers in parallel and simply using the solution provided by the one that finishes first [10]. The solution to the ILP problem shown in Fig. 1 is to assign true to variables t_1, t_2, and t_3. Finally, converting the ILP solution to a minimized test suite is straightforward, since the variables correspond to test cases. Therefore, the approach selects the test cases from T for which the corresponding binary variable is true. The minimized test suite that corresponds to the solution for this problem is T_1, the energy efficient test suite we described in Section II.

IV. Evaluation

To evaluate our approach, we implemented it in a prototype tool, EDTSO, which we used to address the following research questions:

RQ1: Variability Do test cases in a project vary in their energy usage?

RQ2: Potential Impact Do test suites that achieve the same level of coverage vary in their energy usage?

RQ3: Effectiveness Do minimized test suites generated with a focus on energy efficiency use less energy than minimized test suites generated using other approaches?

The purpose of RQ1 and RQ2 is to motivate the need for our approach and the purpose of RQ3 is to demonstrate the feasibility and usefulness of our approach as compared to current techniques.

The remainder of this section describes the prototype implementation of our technique, the experimental subjects we chose, the experimental protocol we used, the data we generated, and the results of our evaluation.

A. Prototype Tool

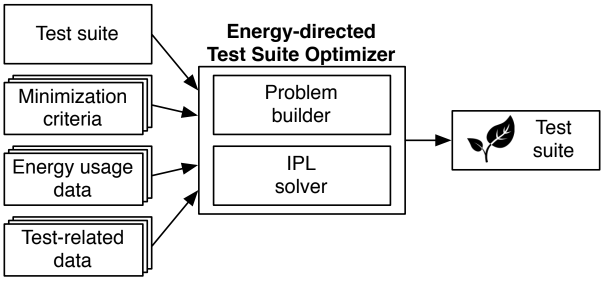

Figure 2. Overview of the the Energy-directed Test Suite Optimizer.

Fig. 2 presents a high-level overview of EDTSO. As the figure shows, EDTSO takes as input a *test suite*, a set of *minimization criteria*, *test-related data*, and *energy data*, and produces a set of green, energy-minimized test suites—subsets of the given test suite that (1) satisfy the given minimization criteria, and (2) use as little energy as possible—one for each platform that energy data is available for.

The *test suite* is the original, full test suite that a developer wants to minimize. In general, the *test-related data* can include many types of information about each test case in the given test suite (e.g., structural coverage information, execution time, manual setup cost, required resources, etc.). However, for our implementation, the data is the instructions covered by each test case. The *energy data* is the energy cost of executing each test case on a platform of interest, in our case, the Low Power Energy Aware Processing (LEAP) node (See Section IV-C2). Note that unlike most other types of test related data, energy data may vary from platform to platform depending on each platform's specific features. The *minimization criteria* are high-level criteria provided by a developer that are used to create the concrete absolute and relative requirements for the minimized test suites. The *problem builder* is responsible for converting the inputs into an ILP problem. See Section III for details on how this is accomplished. And finally, the *ILP solver* is responsible for solving the ILP problem generated by the problem builder. For our experiments, we chose to use lpsolve[2] as our ILP solver.

B. Subjects

We ran our experiments on a set of Android applications that have developer-provided test suites. We chose to consider Android applications for several reasons: First, they are good representatives of the types of applications our approach is designed to target. Android applications typically run on hardware that has a limited power budget (e.g., smartphones and tablets) and interact with sensors that have a high energy

[2] http://lpsolve.sourceforge.net

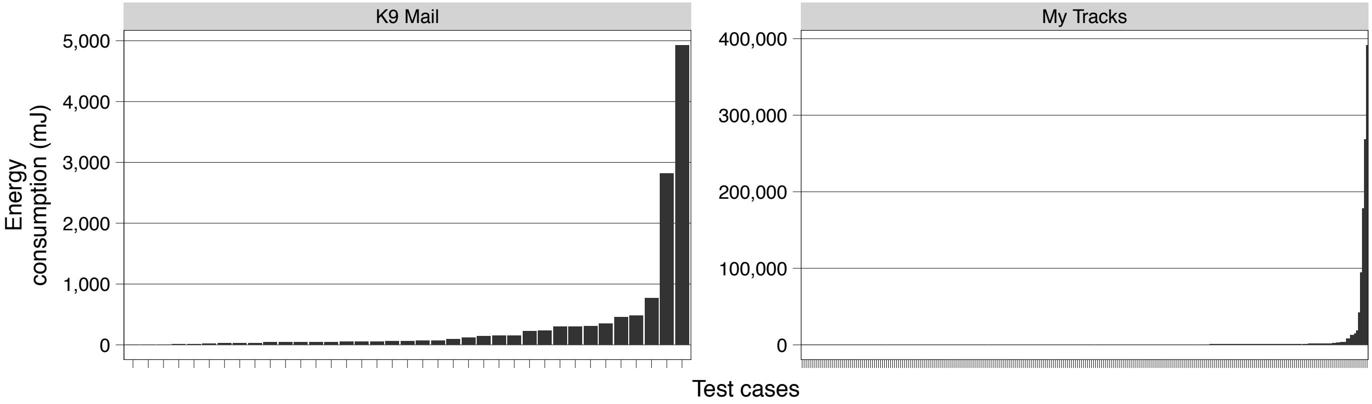

Figure 3. Energy consumption of individual test cases.

Table II
SUBJECT APPLICATIONS.

Subject	Description	LoC	# Tests
K-9 Mail	Email client	71,816	37
MyTracks	Track user paths	35,039	282

cost (e.g., WiFi and GPS). Furthermore, the hardware comes in a wide variety of configurations (e.g., different OS versions, amounts of memory, screen size, etc.), which means that the applications are difficult to adequately test in-house and are an appropriate domain for *in situ* testing. Second, many Android applications are freely available, which allows us to draw from a broad pool of applications subjects that represent different testing and implementation practices. Finally, we have extensive infrastructure, either already developed or available to us, to run Android applications, execute their test suites, isolate the energy consumed by their tests, and collect coverage data.

Table II describes the subjects that we chose to use in the evaluation. The first column in the table, *Subject*, lists the name of each application. The second column, *LoC*, shows the application's number of lines of code and the final column, *# Tests*, shows the number of test cases included in each application's test suite. K-9 Mail is a commonly used, fully featured email client[3] and MyTracks is a popular application for recording the path, speed, and distance a user covers while they are performing an outdoor activity.[4]

C. Experiment Protocol

For each of the subjects, we first collected data about the test cases in each test suite. For each test case, we measured its structural coverage and energy consumption.

1) Structural coverage: To obtain structural coverage data, we instrumented each application using a technique based on efficient path profiling proposed by Ball and Larus [16]. This technique allows us to minimally and efficiently place probes in an application and determine which bytecode instructions were covered during an execution.

2) Energy consumption: To measure the amount of energy consumed by a test case, we ran the instrumented application on a LEAP node [13]. The LEAP is an x86 platform based on an ATOM N550 processor that runs Android 3.2. Each component in the LEAP node (e.g., WiFi, GPS, memory, and CPU) is connected to an analog-to-digital data acquisition (DAQ) card that samples current draw at $10\,\mathrm{kHz}$. The LEAP also provides Android applications with the ability to trigger a synchronization signal. This allows our instrumentation to synchronize the start and stop times of the tests with the samples taken by the DAQ.

The LEAP platform measures energy consumption via hardware external to the smartphone, so it does not introduce any measurement overhead to the application. However, the instrumentation itself has an execution cost, which we subtract from the final energy numbers. To determine the overhead, we profiled the energy cost of the inserted instrumentation. Since we know exactly how many times and when the instrumentation is executed, we can accurately remove its overhead from our energy measurements.

D. RQ1: Variability

To address the first research question, we examined the energy consumption of individual test cases. For each test case in each test suite, we measured its energy consumption using the LEAP node. The test case measurements are shown in Fig. 3. Along the x-axis, the test cases are sorted from left to right in ascending order with respect to their energy consumption. The amount of energy consumed by each test case in milli-Joules (mJ) is shown on the y-axis.

For K-9 Mail, the most expensive test case consumes almost $5{,}000\,\mathrm{mJ}$, the second almost $3{,}000\,\mathrm{mJ}$, with the remainder ranging from approximately $1{,}000\,\mathrm{mJ}$ to just under $5\,\mathrm{mJ}$. Similarly, energy consumption of the test cases for MyTracks ranged from $\approx 5\,\mathrm{mJ}$ to $\approx 400{,}000\,\mathrm{mJ}$. From this data, we conclude that test cases from a test suite are likely to have a high amount of variance in their energy consumption. A certain amount of variance is to be expected, since test cases

[3] https://play.google.com/store/apps/details?id=com.fsck.k9

[4] http://code.google.com/p/mytracks/

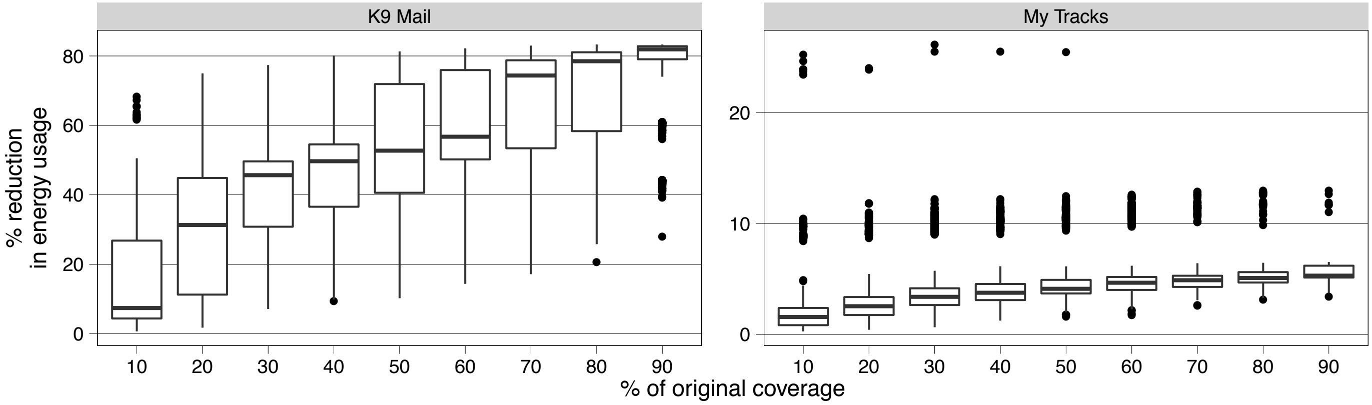

Figure 4. Variability in the energy consumption of minimized test suites.

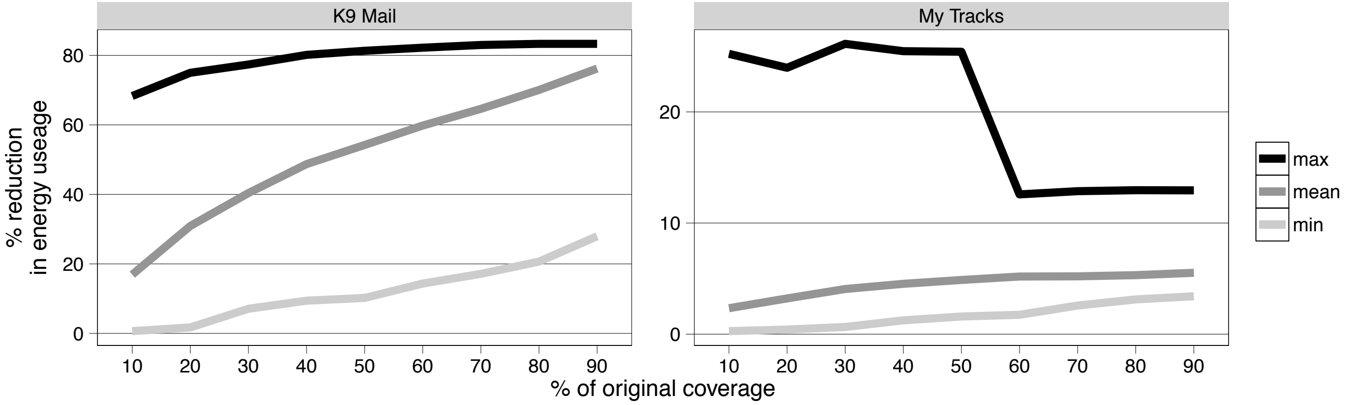

Figure 5. Comparison of the most (top line), average (middle line), and least (bottom line) energy-efficient test suites.

will access expensive system resources in different patterns. However, the amount of variance exhibited in this graph is quite large and suggests that the most common approach for test suite minimization (i.e., selecting the smallest test suite possible) may be sub-optimal when energy consumption is a concern; selecting a larger number of inexpensive test cases is likely to result in a minimized test suite that consumes less energy.

E. RQ2: Potential Impact

To address the second research question, we looked at the variance in energy consumption of regularly-minimized (i.e., without using EDTSO) test suites that achieve the same level of statement coverage. We considered 9 coverage levels (10 %, 20 %, . . . , 90 % of the original test suite's coverage). At each coverage level, we randomly generated 1,000 minimized test suites that achieve the necessary coverage. We then compared the energy consumption of the randomly generated test suites by normalizing them against the energy consumption of the original test suite.

Fig. 4 shows a box plot that summaries the result of this experiment. Along the x-axis are the different levels of coverage that we considered. The y-axis shows the amount of energy used as compared to the application's original test suite. In each box, the horizontal line and the upper and lower edges show the median and the upper and lower quartiles, respectively for the percentage of energy used at each coverage level.

As this graph shows, for a given level of coverage, there is a wide range in the amount of energy consumed by the test suites. To highlight this trend, Fig. 5 shows a plot of the maximum, minimum, and average energy consumption values observed for each of the 1,000 test suites generated at each coverage level. This graph shows that the most energy inefficient test suite can consume much more energy than the most energy-efficient test suite, in many cases, the best choice to satisfy a coverage level is 30 % to 70 % more efficient than the worst choice. Taken together with the results for RQ1, these two experiments provide a compelling argument for the use of energy oriented test suite minimization by showing that less test cases might not be better and that there is a significant difference in energy consumption of test suites even though they achieve the same coverage level.

F. RQ3: Effectiveness

To address the third research question, we considered two metrics, the amount of energy savings achieved by EDTSO and the time needed to perform the minimization.

Table III
IMPROVEMENT IN ENERGY CONSUMPTION FOR MY TRACKS.

	% Improvement in Energy Usage					
Coverage (%)	Min.	Q_1	Median	Mean	Q_3	Max.
10	0	33	60	55	77	100
20	0	26	43	45	64	98
30	1	22	36	39	50	98
40	1	19	32	35	46	95
50	1	20	30	34	42	78
60	2	22	28	31	38	77
70	2	23	27	30	37	75
80	8	22	25	28	35	73
90	8	21	24	27	35	70
100	34	34	34	34	34	34
Overall	0	22	31	36	44	100

Table IV
IMPROVEMENT IN ENERGY CONSUMPTION FOR K-9 MAIL.

	% Improvement in Energy Usage					
Coverage (%)	Min.	Q_1	Median	Mean	Q_3	Max.
10	0	0	0	10	0	98
20	0	0	0	20	36	92
30	0	0	0	20	39	85
40	0	0	0	16	32	94
50	0	0	0	11	3	73
60	0	0	0	8	0	71
70	0	0	0	4	0	60
80	0	0	0	2	0	57
90	0	0	0	1	0	49
100	0	0	0	0	0	0
Overall	0	0	0	10	0	98

First, we compared the energy consumption of test suites minimized using our energy-oriented approach against traditionally minimized test suites. Similar to our procedure for investigating RQ2, we generated 1,000 minimized test suites for each coverage level using both EDTSO and a traditional minimization approach that focused only on test suite size. We then performed a pair-wise comparison of the energy usage of the minimized test suites generated by each approach and calculated the percentage improvement in energy consumption achieved by EDTSO.

Tables III and IV show a summary of this comparison. For each coverage level, the tables show the minimum, first quartile, median, mean, third quartile, and maximum values of the pair-wise improvement comparisons. Note that all values are rounded to the closest integer and there were no negative numbers reported.

As the tables show, in the majority of cases, EDTSO generated more energy-efficient minimized test suites and, in addition, never created a test suite that consumed more energy than the corresponding test suite generated by the traditional approach. In practice, this means that developers who use EDTSO will have a test suite that is at least as good as they could have created otherwise and at best a test suite that is significantly more energy efficient. In essence, using EDTSO places a developer in a no-lose situation.

For My Tracks, the results of the experiment are especially promising; EDTSO created a better test suite 100 % of time with the amount of improvement ranging from $\approx 1\,\%$ to 99 % with an average of $\approx 36\,\%$. For K-9 Mail, the results are not as strong; EDTSO created a better test suite approximately 22 % of the time with the improvement ranging from $\approx 1\,\%$ to 98 % with an average of $\approx 48\,\%$. However, we believe that this relatively poor performance is due primarily to the limited number of test cases for K-9 Mail. Often statements are only covered by one test case; in such situations both approaches must make the same choice, so there is no possibility for improvement.

Second, we investigated the amount of time needed by EDTSO to produce energy-efficient, minimized test suites. For all of the 18,002 minimized test suites that we considered (1,000 test suites for each coverage level from 10 % to 90 % plus 1 test suite for the 100 % coverage level times 2 applications), EDTSO was able to generate an energy-efficient, minimized test suite in under 5 s. This result is also promising as the speed of EDTSO will allow it to be run frequently as part of the normal development cycle.

V. RELATED WORK

Unlike our technique, which minimizes a test suite with respect to the current version of a program, the majority of existing work on test case selection and minimization is defined to operate in the context of regression testing (i.e., testing a new version of a program based on information gained from analyzing previous versions). At a high-level, such techniques can be divided into several categories.

Existing work on test suite minimization, also known as test case selection, can be divided into several high-level categories [17]: *Retest all techniques*, which simply rerun all existing test cases. *Random or ad hoc techniques*, which randomly select an arbitrary percentage of the existing test cases. *Minimization techniques*, which select the minimum number of test cases necessary to achieve a given criterion. For example, a minimization technique could select the smallest number of test cases such that all added or modified statements in the new version are executed. *Safe techniques*, which select every existing test case that achieves a certain criterion. For example, a safe technique could select every test case that executes an added or modified statement in the new program. Yoo and Harman provide a recent, detailed survey of existing work in each of these categories [18].

Because the types of techniques mentioned above are designed for regression testing, they should not be seen as alternative to our approach, but rather complimentary. Incorporating our approach with existing regression techniques would allow these approaches to also consider energy requirements in their application.

In addition to test suite minimization techniques, our approach is also related to work on post deployment validation, or more generally, techniques that leverage data collected from deployed software. Like the test suite minimization techniques described above, techniques that gather information from the field are not alternatives to our technique. Instead, they represent opportunities where our general approach of optimizing with respect to energy usage can be applied.

The software engineering community has proposed a significant number of techniques. Most relevant to our work are techniques proposed by Pavlopoulou and Young [5] and Murphy and colleagues [6], [19]. These techniques advocate continuing the testing process after an application is deployed by embedding portions of a test suite inside of an application. In addition to techniques that attempt to validate software after deployment, there are also techniques that help improve: debugging (e.g., [7]), fault prediction and detection (e.g., [20]), effort analysis (e.g., [21]), and performance analysis (e.g., [22]).

Currently, none of these approaches take into account the energy consumption of the test cases. We believe that by incorporating energy consumption requirements into these approaches, their energy consumption could also be reduced.

VI. Conclusions and Future Work

In this paper we presented a new test suite minimization technique for generating test suites with reduced energy consumption. Our approach is based on casting the energy and test coverage requirements as an ILP problem. We implemented our approach in a prototype tool and used it to conduct an empirical evaluation using the test suites provided for two Android applications. In the evaluation we found that the energy consumption of individual test cases varied significantly within a test suite, and that even test suites with identical coverage could vary in energy consumption by 30 % to 70 %. This result strongly motivates the need for test minimization techniques, such as ours, that focus on energy minimization. Our evaluation also found that compared to traditional test suite minimization techniques, our approach could identify test suites that maintained coverage while reducing energy consumption an average of 5 % to 48 %. Overall, the results were very positive and indicate our technique can help developers of mobile and embedded software reduce the energy consumption of their test suites.

Although the results of our evaluation were positive, we consider our work ongoing and have identified several areas of future work. First, we would like to perform more extensive empirical evaluations with a wider variety of applications and larger test suites to improve the generalizability of our results. Second, in the evaluation, we would like to compare more attributes of the test suites, such as execution time and number of API invocations, to better understand the factors that affect test case energy consumption. Finally, we would like to refine our approach to allow for more flexible specification of the energy requirements.

References

[1] K. Langendoen, A. Baggio, and O. Visser, “Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture,” in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, 2006, pp. 174–174.

[2] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, “The hitchhiker’s guide to successful wireless sensor network deployments,” in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, 2008, pp. 43–56.

[3] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, “A macroscope in the redwoods,” in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, 2005, pp. 51–63.

[4] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, “Fidelity and yield in a volcano monitoring sensor network,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 381–396.

[5] C. Pavlopoulou and M. Young, “Residual test coverage monitoring,” in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 277–284.

[6] C. Murphy, G. Kaiser, I. Vo, and M. Chu, “Quality assurance of software applications using the in vivo testing approach,” in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 111–120.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 15–26.

[8] Z. Chen and K. G. Shin, “Post-deployment performance debugging in wireless sensor networks,” in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 313–322.

[9] J. Black, E. Melachrinoudis, and D. Kaeli, “Bi-criteria models for all-uses test suite reduction,” in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE ’04, 2004, pp. 106–115.

[10] H.-Y. Hsu and A. Orso, “MINTS: A general framework and tool for supporting test-suite minimization,” in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 419–429.

[11] G. Rothermel and M. J. Harrold, “A safe, efficient regression test selection technique,” *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, 1997.

[12] M. J. Harrold, R. Gupta, and M. L. Soffa, “A methodology for controlling the size of a test suite,” *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, 1993.

[13] 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, http://lasr.cs.ucla.edu/leap/FrontPage?action=AttachFile&do=get&target=leapwhitepaper.pdf.

[14] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, “Estimating mobile application energy consumption using program analysis,” in *Proceedings of the 35th International Conference on Software Engineering*, 2013, to appear.

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

[16] T. Ball and J. R. Larus, “Efficient path profiling,” in *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, 1996, pp. 46–57.

[17] G. Rothermel and M. J. Harrold, “Analyzing regression test selection techniques,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 529–551, 1996.

[18] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, 2012.

[19] C. Murphy, M. Vaughan, W. Ilahi, and G. Kaiser, “Automatic detection of previously-unseen application states for deployment environment testing and analysis,” in *Proceedings of the 5th Workshop on Automation of Software Test*, 2010, pp. 16–23.

[20] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “Cross-project defect prediction: A large scale experiment on data vs. domain vs. process,” in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2009, pp. 91–100.

[21] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, “Local vs. global models for effort estimation and defect prediction,” in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 343–351.

[22] A. Bertolino, G. De Angelis, and A. Sabetta, “VCR: Virtual capture and replay for performance testing,” in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 399–402.