COMPUTING AND DIAGNOSING CHANGES IN UNIT TEST ENERGY CONSUMPTION

Michal Young
Jamie Andrews
Andrew J. Ko
Brian P. Robinson
Mark Grechanik

introduction

Many developers have reason to be concerned with with power consumption. For example, mobile app developers want to minimize how much power their applications draw, while still providing useful functionality. However, developers have few tools to get feedback about changes to their application's power consumption behavior as they implement an application and make changes to it over time.

We present a tool that, using a team's existing test cases, performs repeated measurements of energy consumption based on instructions executed, objects generated, and blocking latency, generating a distribution of energy use estimates for each test run, recording these distributions in a time series of distributions over time. Then, when these distributions change substantially, we inform the developer of this change, and offer them diagnostic information about the elements of their code potentially responsible for the change and the inputs responsible. Through this information, we believe that developers will be better enabled to relate recent changes in their code to changes in energy consumption, enabling them to better incorporate changes in software energy consumption into their software evolution decisions.

To understand the scenario for the tool, imagine a developer changes a case insensitive string comparison to case sensitive string comparison, saves, and one unit test reports that there was a 20% increase in energy consumption. The developer hovers over the feedback and the tool highlights the offending calls, showing the developer that it was a particular call of a function that contained that new string comparison that is being called repeatedly. In this case, the developer can weigh the decision of using the case sensitive comparison with increased power or using the case insensitive comparison while keeping power low. Without this data, the developer would not have known about this potential change.

Contributions are reporting significant changes in power consumption behavior from unit test analysis and in reporting which elements of the unit tests are likely to be responsible for the changes.

related work

Power consumption research in software is a recent topic, but there are still many approaches.

For example, Chan et al. [Chan 2009] contribute an approach to test definition that looks for deviations in power consumption in wireless sensor networks, while also accounting for the power consumption involved in actually executing tests. This approach is tailored specifically for wireless sensor nodes.

Transmeta’s Code Morphing technology [Klaiber 2000] changes the entire approach to designing microprocessors. By demonstrating that practical microprocessors can be implemented as hardware-software hybrids, Transmeta has dramatically expanded the design space that microprocessor designers can explore for optimum solutions.

Dagstuhl Seminar Proceedings 10111
Practical Software Testing : Tool Automation and Human Factors
http://drops.dagstuhl.de/opus/volltexte/2010/2624

The Efficeon processor is Transmeta's second-generation 256-bit VLIW design which employs a software engine to convert code written for x86 processors to the native instruction set of the chip (Code Morphing Software, aka CMS). Like its predecessor, the Transmeta Crusoe (a 128-bit VLIW architecture), Efficeon stresses computational efficiency, low power consumption, and a low thermal footprint.
http://en.wikipedia.org/wiki/Efficeon

Girard et al's work instructs readers how low-power devices can be tested safely without affecting yield and reliability [Girard et al. 2010]. Includes necessary background information on design for test and low-power design. Incorporates detailed coverage of all levels of abstraction for power-aware testing of (low-power) circuits and systems.

There are also several other power consumption optimizations, including energy aware compilers and approaches for scheduling and laying out distributed systems. There are also reference testing approaches for the power consumption of devices, in which we compare consumption on multiple devices.

measuring a test's energy consumption

To measure energy consumption, we will use existing energy models for a processor, counting instruction executions by type, objects generated to account for garbage collection, and time blocked by interactions with server requests and other external entities. We will repeatedly run a test to sample energy consumption from non-deterministic behavior, like blocking and threads, producing a distribution of energy consumption measurements.

diagnosing the cause of changes to tests' energy use

Our approach is applicable when an increase in energy use between successive versions of a program is unevenly distributed across program executions, that is, when we can distinguish *good* executions (normal or decreased energy use) from *bad* executions (increased energy use). The approach comprises two steps. In the first step, pairs of executions (old and new) are classified. The classification step can be considered a form of regression testing. In the second step, executions drawn from two contrasting buckets are analyzed to *allege blame* on portions of the program.

Note that the distinction between *bad* and *good* is not based on energy use in the current version only, but rather relates executions that were expected to exhibit like behavior (modulo intentionally observable changes in program functionality). Different definitions of "bad" are possible. For example, we might characterize as "bad" those executions whose energy use is at least 50% greater than in an otherwise similar execution of the previous version, or those whose execution is two standard deviations greater than *n* standard deviations from the prior version.

In addition to *good* and *bad* executions, two more classes may be distinguished. Since there is typically some intended change from one version to the next, some execution pairs may be deemed incomparable, because they differ sufficiently in other observable characteristics (amount of output produced, effects on resources outside the program, etc.) that one should not expect their energy consumption to be similar. In addition, it is desirable that comparable execution pairs fall very clearly into either the *good* or the *bad* bucket; it may be necessary to add an intermediate *semi-bad* classification between them. Execution pairs from the *semi-bad* bucket are discarded and have no further use.

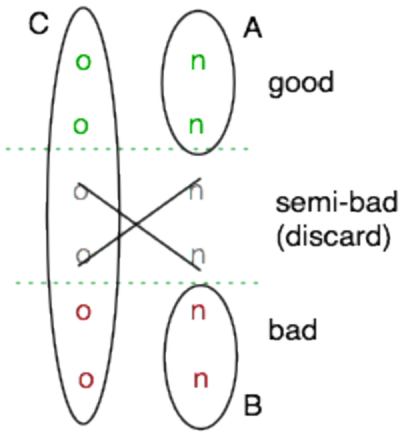

Figure x: Step 1 (bucketing) distinguishes "good" execution pairs from "bad", comparing old (o) and new (n) versions of the software. Step 2 compares execution characteristics (dynamically detected invariants) of executions of the new version, drawn in equal number from the good and bad buckets. Characteristics of interest are those that appear in C (they were always true in the previous version of the system), and also in A (they are still true in the "good" executions), but not in B (the bad executions deviate from the old and good executions).

The second step begins with re-execution of the new components of pairs from the bad bucket, and an equal number of execution pairs taken from the good bucket (groups "A" and "B" in Figure x). In this re-execution, a dynamic invariant detection tool such as Daikon is used to gather information that characterizes the "good" executions as a group, and separately to gather information that characterizes the "bad" executions as a group. The number of samples from the two buckets is balanced so that putative invariants will not appear in only one of the two sets simply because of differences in statistical confidence. If the buckets are of sufficient size, the samples may be balanced in other regards as well, although in general we should not be surprised to find systematic differences between the two buckets, and this is not a problem so long as the differences are indeed representative rather than an accident of sampling. Energy use is not measured in the second step, as it is certain to be perturbed by monitoring for invariant detection.

When invariants of the old version of the software (group "C" in Figure x) are compared to invariants of the good new executions and invariants of the bad new executions, the question we ask is: Are there invariants of the old system that were preserved in the good executions, but violated in the bad new executions? Since invariants detected by a tool like Daikon are specific to points in the source program (typically procedure entry and exit), we can consider each such change as an *allegation of blame*.

True blame may not lie exactly where the allegation points. If an invariant at procedure entry has changed, then blame could lie with the calling procedure; called procedures may also deserve blame if their entry invariants are preserved but invariants at exit are changed. Red herrings (changes to invariants that are highly correlated with differences in energy use, but not to blame for it) are also possible, but even they may provide useful hints as to the actual cause of increased energy use.

a regression approach

We also propose learning regression models of power consumption from performance data. Given the model derived from the baseline version of the program, we evaluate how accurate the model is on the new version of the program, and how the model would have to change to conform better to the new version. We then present the developer with the differences in power consumption between the baseline and new versions of the program when run on the test suite, and the differences in the regression models. We then study the hypothesis that this data is considered useful by the developer.

Given source functions $f_1, \ldots, f_n$, we log each call to each f_i. Given test case T_j, for j from 1 to m, we record the number of times function f_i has been called in variable c_{ij}. We then search for a model linking the c_{ij} values (or common functions on those values) to the
power consumption P_j of test case T_j.

We view this model search as an instance of the {\it model selection} problem. That is, given the {\it predictor} variables \\
$c_1, log(c_1), c_1 \cdot log(c_1), c^2_1, \\
 c_2, log(c_2), c_2 \cdot log(c_2), c^2_2, \\
 \ldots, \\
 c_n, log(c_n), c_n \cdot log(c_n), c^2_n$, \\
and the {\it outcome} variable P, we seek a subset $x_1, \ldots, x_p$ of the predictor variables
and corresponding coefficients $a_1, \ldots, a_p$ such that the following is true for all test cases T_j:
\[P_j \cong a_1 \cdot x_{1j} + \cdots + a_p \cdot x_{pj} \]

In order to do this, we apply the standard model selection procedure LARS \cite{efron-etal-lars-2004}, as implemented in the R statistical package \cite{venables-etal-R}. This procedure adds predictor variables one by one while building a linear regression model, until some desired measure of accuracy of the model has been achieved. We take as our desired measure of accuracy an adjusted R^2 value of 0.95. (R^2 is the square of the standard Pearson correlation; adjusted R^2 is a measure
that compensates for the number of variables in the model; and 0.95 is considered a high level of linear regression model accuracy.)

The result is an equation that links power consumption to the number of times given functions in the source code have been called, or the logs, squares and so on of those numbers. The use of logs, squares and so on makes it more likely that if there is a non-linear relationship between the predictor and outcome variables, the LARS procedure will be able to find it. The use of the threshold value of 0.95 guards against overfitting and makes it more likely that a relatively small number of variables will be involved in the model (we predict between one and eight variables).

One such model may or may not be informative to the developer. However, when comparing a model of power consumption of a baseline version to the model of power consumption of a new version, we hypothesize that two classes of changes may indeed be interesting to developers.

First, if a coefficient a_k has increased (resp.\ decreased) substantially, this may indicate that the corresponding function is taking more (resp.\ less) power to compute in the new version, when compared to the baseline. It would be up to the developer to decide whether the new version is taking too much power, and whether the increase or decrease in power

consumption of the function
is acceptable.

Second, if a predictor variable has been added or deleted in the new model when compared to the baseline model, this may indicate that the corresponding function has newly become an important
influence on the power consumption of the application. Again, it would be up to the developer to decide whether this is a problem.

We hypothesize that the first few iterations of the model-building process will not be very informative to the developer, but that as they inspect the historical trend in the models over various different releases, they will get a better intuitive sense of how to use and draw conclusions from the
models.

presenting energy use changes to developers

Once we have computed the energy use changes in tests and predicted the cause of these changes, we then present these changes to developers. Our assumption is that tests would already be divided into some meaningfully distinct situations: for example, one test might regard a login procedure and another might test a save feature. Therefore, we will present change information at the level of a test.

We will present information as proportional changes. For example, a unit test will be described as having increased in energy consumption by an approximate percentage. We do this instead of other more user-centered units (such as minutes of battery life lost) because the system will have no knowledge of the representativeness of tests with respect to the application's intended use. Providing the proportional change allows a developer to make this representativeness assessment with their domain knowledge and intents in mind.

evaluation

Any good tool evaluation will correspond to claims. Therefore, our claims are:

1) developers find these deltas useful in making energy consumptions decisions
2) our approaches for generating deltas, including the Daikon approach and the coefficient and predictor variable approaches, is found to be meaningful by developers
3) developers can interpret the proportional changes in a meaningful and accurate way
4) developers can use the partial diagnostic information to perform a full diagnosis
5) the developer considers the information provided by the models to be more useful the longer they observe it.

Given these claims, we would provide a lab study in which developers would make a series of modifications to a program, observing the system's predictions of each test's energy use change, and verbalizing their interpretations of the meaning of the predictions. We would also ask participants to find the cause of the change in each test, observing to what extent the diagnostic information would aid developers in selecting code elements likely responsible for the change. Our tool would only provide value if developers were making valid and accurate interpretations of the system's predictions; this study would allow us to assess this.

bibliography

"The Daikon system for dynamic detection of likely invariants"

by Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao.
Science of Computer Programming, vol. 69, no. 1--3, Dec. 2007, pp. 35-45.

"Dynamically discovering likely program invariants to support program evolution" by Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin.
IEEE Transactions on Software Engineering, vol. 27, no. 2, Feb. 2001, pp. 99-123.

"Towards the Testing of Power-Aware Software Applications for Wireless Sensor Networks." W. K. Chan, T.Y. Chen, S. C. Cheung, T.H. Tse, and Zhenyu Zhang. In N. Abdennahder, F. Kordon (Eds.): *Proc. Ada-Europe 2007*, LNCS 4498, pp. 84–99, 2007. Springer-Verlag Berlin Heidelberg 2007.

"The Technology Behind Crusoe Processors", Alexander Klaiber. *Transmeta Corporation*, January 2000.

"Power-Aware Testing and Test Strategies for Low Power Devices". Girard, Patrick; Nicolici, Nicola; Wen, Xiaoqing (Eds.). 1st Edition., 2010, XXII, 353 p. 444 illus., 222 in color., Hardcover. ISBN: 978-1-4419-0927-5

"Least angle regression". B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. *Annals of Statistics*, 32(2):407–499, 2004.

"An introduction to R." W. N. Venables, D. M. Smith, and The R Development Core Team. Technical report, *R Development Core Team*, June 2006.

THOUGHTS FROM EARLIER (unrelated to the paper above)

ideas

developers label automatically generated assertions good or bad, test suite is regenerated based on these labels, ultimately reducing test oracle brittleness.

apply delta debugging to minimize assertion sequence, such that test still fails.

people perform normal tasks on an application, labeling "slow" scenarios, while capturing call profiles. From these labels and call profiles, we generate unit level performance tests.

simplifying assumptions

assume an incentive an for labeling
assume a meaningful test, something understandable
assume some understanding of the code

what to do with labels

The mental task of assessing the correctness of an assertion is quite demanding, so they better get something meaningful from it. So what we do with that label better be pretty useful!

Michal wants to use the label to have to ask for labels less often.

It's **much** easier to label concrete executions because developers have a rich understanding of the meaning of the execution. It's inherently more difficult to label units because they're inherently more abstract. Therefore, part of the challenge of eliciting labels is finding some more concrete scenario for a developer to label and map that concrete scenario to more abstract test cases. **What types of mapping mechanisms might we use? Machine learning? Program analyses?**

Another way to make things more concrete is to better visualize the data that's being manipulated, to facilitate oracle judgements. For example, showing a sorting algorithm run visually, instead of watching the code execute it, is going to better facilitate correctness judgements. **Are there ways to generate these visualizations automatically? Are there domains for which we can design these visualizations? Does any of this generalize beyond a particular application?**

non-functional qualities that might have low hanging fruit

variable name comprehensibility
performance
latency
echo latency/responsiveness
confusing labels on a web form
legibility
feedback proximity
contrast in visual semantics is conveying meaning
testability (observability and controllability)
security (no expert in the room!)
compliance (could be arbitrary requirements for data with certain properties to be computed)

data loss (all values with human origin must find a home by the end of a test)
consistency (many interpretations; for example, browser compatibility)
standards compliance (acid3 for HTML and CSS)
energy consumption

optimizations

17 optimizations the compiler was able to do are no longer applicable.