Interpreting Energy Profiles with CEGAR

Steven te Brinke
University of Twente – Formal Methods and Tools group – Enschede, The Netherlands
brinkes@cs.utwente.nl

ABSTRACT

Decreasing the energy consumed by systems that are controlled by software is important. To facilitate modular implementation of energy optimization logic, we model energy consumption of software as Resource-Utilization Models (RUMs). The CEGAR approach can automatically extract RUMs from existing component implementations, but it cannot measure energy consumption. Therefore, we show how CEGAR can be augmented with profiling to automatically interpret energy profiles, so as to extract RUMs containing energy information without human intervention.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*

General Terms

Design, Performance

Keywords

energy-aware software, modularity, CEGAR, model checking, resource-utilization model, minimal abstraction

1. INTRODUCTION

There is an increasing demand for reducing the energy consumption of systems that are controlled by software. Energy is one of the resources that should be reduced, but since software often consumes higher-level resources which indirectly consume energy, it is important to model not only energy, but also resource consumption in general. To facilitate modular implementation of resource optimization logic, we have proposed [5] to use so-called *Resource-Utilization Models* (RUMs), which express the relation between the dynamic behavior of the component and the resources it uses and provides as state transition diagrams expressing transitions—triggered by either service invocations or internal events—between states of stable resource consumption. In our previous work, we have also shown how to use the CEGAR approach to automatically extract RUMs from existing component implementations.

However, this approach does not measure any energy consumption; it assumes that energy information is available already, e.g.: as annotations in the source code or defined by the specification. Whereas this assumption holds in some cases, it is not applicable in general: software libraries usually lack energy information. Therefore, to optimize energy consumption effectively, it is necessary that the energy consumption of such libraries can be profiled, so as to add energy information to the RUM.

2. BACKGROUND

Our approach—as presented in previous work—uses CEGAR to extract RUMs from source code. CEGAR assumes that source code contains enough information to create a model of its energy consumption. Nevertheless, in general, source code does not contain energy information, so an additional source of information is needed. Therefore, this paper describes how we add profiling to our approach.

2.1 Profiling

Profiling the software at hand is a way to acquire energy information without manually specifying its energy behavior during development. Profiling executes the software and measures its actual energy consumption. Each run profiles a single execution sequence, so multiple runs are required for extracting an energy profile. By combining profiling with CEGAR, it is possible to extract RUMs automatically from source code that lacks energy information.

2.2 CEGAR

Counterexample-guided abstraction refinement (CEGAR) is a formal method to refine abstract models when a concrete model is available. CEGAR consists of the four steps illustrated in figure 1. First, an initial abstract model is derived by static analysis of the source code. Second, the desired energy consumption is checked against the abstract model. When model checking cannot prove a given energy property on the abstract model, a counterexample is produced. Third, the counterexample is simulated on the concrete model, which may have two outcomes:

- There is *no* real error, but the abstract model does not include enough information about the concrete behavior; it is called a *spurious* counterexample.
- There *is* a real error. Then an inventive step is needed, because either the application misbehaves or, more likely, the property is a too strict requirement.

In the first case, step four can automatically extract information from the concrete model to make a refined abstract model in which the previous counterexample cannot occur, and then this process must be repeated until the desired property can be proven on the refined abstract model.

In the second case, when we discover that the property is too strict, we should relax the requirements on energy consumption. This can be guided by the counterexample, but is not automatic.

3. EXAMPLE AND REQUIREMENTS

We will now give an example that demonstrates typical steps in CEGAR when combined with profiling. The state

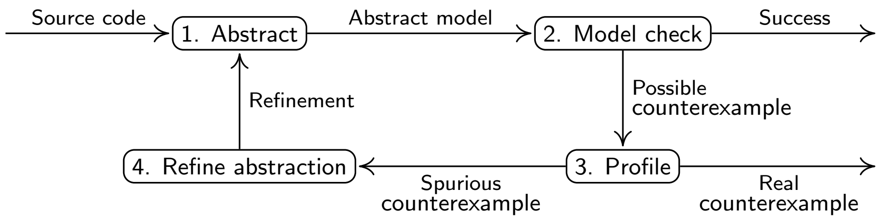


Figure 1: Steps of CEGAR

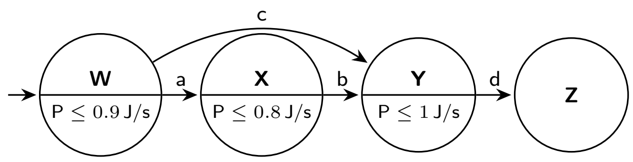


Figure 2: Example behavior

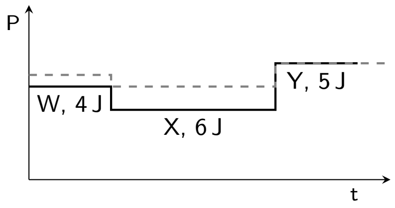


Figure 3: Actual energy consumption

chart in figure 2 is the behavior (i.e.: RUM) of an application on which we run our approach. Deriving this RUM started with specifying the key property: *reaching Z always consumes less than 16 J*, which cannot be guaranteed. Therefore, our approach will give a counterexample, which could be the following sequence that consumes 17.5 J (denoted by which states are visited, how long the application stays in these states, and how much energy is consumed in each state): W, 5 s, 4.5 J $\rightarrow$ X, 10 s, 8 J $\rightarrow$ Y, 5 s, 5 J $\rightarrow$ Z.

We must validate that this counterexample is spurious and derive all necessary information missing in the RUM from the application. From the state chart, we see which services must be invoked to visit the states of the counterexample. We assume that we can invoke the services a, b, and d at the moments given by the counterexample. This allows us to generate a test case that stays in each state as long as given by the counterexample by executing these services at the desired times. Executing this test shows us whether both the timing and the energy consumption of the counterexample are realistic. Figure 3 depicts a possible output profile for this example: it shows the measured energy consumption over time. We see that in reality, the counterexample did not occur, because both states W and X consume less energy than the maximum allowed by the RUM in figure 2. Thus, we can be confident that the counterexample is spurious and we refine the RUM with the actual energy consumption. On the refined RUM, the above steps are applied again iteratively, which shows that the key property holds.

3.1 Requirements

Profiling does not provide energy information for every possible execution sequence of the application. However, CEGAR expects an *over*-abstraction as the concrete model, which contains at least every execution sequence. Also, energy consumption might be dependent on state outside the application that is being profiled, which is invisible to the profiler. For example, the state of the network driver can influence the energy needed for transferring data over the network. Thus, the energy consumption of a specific execution sequence cannot be acquired exactly with a single run only: we must take into account that profiles are inexact, whereas CEGAR expects an exact specification (an over-abstraction) as concrete model.

Therefore, our challenge is *(a)* combining two models: source code plus profiled energy consumption, and *(b)* adapting CEGAR to work with uncertain (or probabilistic) data.

3.2 Goals of Extracting Energy Profiles

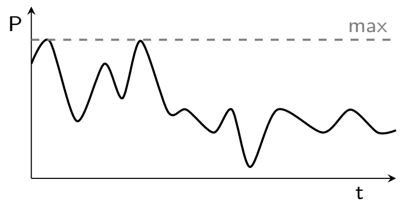
Our goal is extracting exact energy profiles that are usable by CEGAR from inexact information provided by profiling. In reality, we expect the energy consumption to be more dynamic than shown in figure 3, for example as shown by the graphs in figure 4. Extracting a precise profile may require correlating changes in energy consumption to execution events, such as service invocation. The dashed *max* lines in figure 4 show three possible energy profiles which can be extracted from the same measurement.

The more precisely we can extract the energy profile, the more useful the result will be. We define precision as the difference between the energy profile p and actual energy consumption a: $\int |p(t) - a(t)| dt$, where lower is more precise. (Note that when we profile the maximum, $p(t) \geq a(t)$ must hold, so we can omit taking the absolute value.) In figure 5, the precision is shown as the gray area between the actual consumption (the solid black line) and the profile for the maximum energy consumption (the dashed gray line).

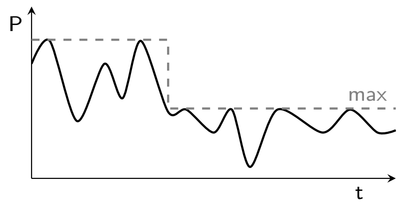
Figure 4a shows the simplest profile: the energy consumption is always below a constant value. However, this value is often much higher than the actual energy consumption, so it is not a very precise profile.

In figure 4b, the maximum is reduced at some moment. This creates a more precise profile, but requires knowledge about when the maximum energy consumption changes (e.g.: after a certain time or after invocation of a specific service).

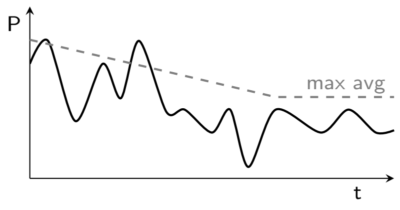
Instead of providing a maximum for the consumption at any moment, figure 4c provides a maximum for the average energy consumption. Since, in general, we are interested in the total energy consumption, having guarantees on the average consumption is sufficient. Even though an energy profile of the average consumption provides less information than a profile of the energy consumption at any moment, such a profile can be useful when it provides higher precision. Figure 5c shows the actual average as dotted line and the precision as the gray area between this line and the profile. We see that energy profile 3 provides highest precision.



(a) Energy profile 1

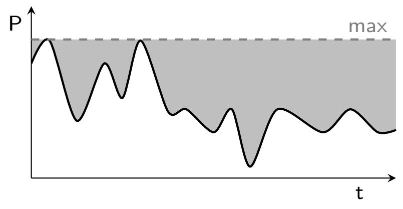


(b) Energy profile 2

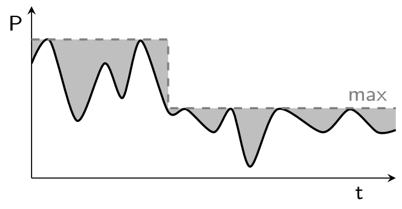


(c) Energy profile 3

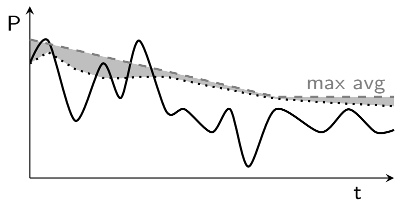
Figure 4: Example energy profiles



(a) Energy profile 1



(b) Energy profile 2



(c) Energy profile 3

Figure 5: Precision of energy profiles

3.3 Goals of Uncertainty in CEGAR

In section 2.2, we consider a counterexample spurious if it *does not* occur in reality. When adding uncertainty to CEGAR, we consider a counterexample spurious if *we are confident that it will not* occur in reality. Thus, there still is a possibility that a spurious counterexample will occur eventually. Profiling cannot proof the absence of counterexamples, it can merely find counterexamples for the counterexamples. This way, we can gain confidence that the counterexample cannot occur in reality, but we can never proof so.

For now, we manually decide whether we have enough confidence. Automatically deriving the confidence from a set of profiles is still future work.

4. CEGAR WITH PROFILING

Existing tools can already perform CEGAR, but none of these tools provide the ability to include the results of profiling. To add profiling to the CEGAR process, we created our own tool chain, which will be explained in this section.

4.1 Abstracting

The CEGAR process starts with extracting an initial abstraction from code. In our case, we use Java bytecode as source code, and extract a model from it with our Java Bytecode ++ (jbcpp) tool[1]. The initial jbcpp model is quite large, since it contains *all* bytecodes that are present in the program. Because we are interested in an abstraction, we group sequences of bytecodes, as long as these do not contain any of the following:

1. Method calls
2. Branching

We would like to extract the portion of the model that consumes significant power, because that is what we are going to profile. A single bytecode operation is too short to contain significant power, therefore only a long sequence of such operations should be considered. Method calls are represented by a single bytecode instruction, but do invoke behavior specified elsewhere, which may consume significant power. Therefore, method calls are kept.

Branching influences the path that will be taken, which can be a choice between high and low power consumption. Also, loops are represented as branching and repeated execution may result in significant power consumption. Therefore, CEGAR is interested in conditionals, such that it can decide which of the paths will be executed. Thus, branching is also kept during abstraction.

Initially, little is known about the energy consumption of the application. The initial abstraction will therefore assume that the application might consume its maximum power consumption at any moment in time. This leads to an over-approximation of the power consumption, which will be refined by CEGAR later.

4.2 Model checking

This step verifies whether the given key property holds on the abstract model. In our tool chain, we model check with UPPAAL [4], by transforming the jbcpp models to UPPAAL models. These UPPAAL models are then used to check key properties, for which UPPAAL can generate traces representing abstract counterexamples. Such counterexamples are traces that are interesting to profile.

As explained in section 2.2, the abstract counterexamples are either *(a)* spurious counterexamples because not enough information is known about the concrete behavior, or *(b)* real errors that should be fixed by the responsible developer or architect. The goal of profiling is identifying which of these is the case and acquiring more information when the former is the case. Thus, the counterexamples provided by UPPAAL are the traces that we profile.

4.3 Profiling

For profiling, we use the traces acquired as counterexamples during model checking. However, we cannot profile every instruction present in a trace, because the overhead of profiling is only negligible when the profiled part consumes a considerable amount of time and resources. Since we are

[1]See the SourceForge project *jbcpp* at `jbcpp.sf.net`.

trying to create an abstraction that provides bounds on the execution times and resource consumption, we are also only interested in parts consuming considerable time or resources.

The goal of profiling is to acquire an energy profile that is detailed enough to prove the given key properties. Because we cannot profile every instruction, we have to select ranges that are most likely useful to profile. Since profiling can be automated, it is not needed that every acquired profile provides useful information, but the CEGAR loop must be able to reach useful profiles automatically.

We choose loops as starting points for inserting profiling code, as in general it is hard to predict how long a loop will run and how much energy it will consume. Therefore, an over-abstraction might initially assume that both runtime and thus also energy consumption are infinite. However, in practice the runtime is bounded, which also bounds the energy consumption.

Thus, we start with profiling the outermost loop present in the counterexample to see if we can acquire energy bounds that satisfy the given key property. Note that we are not interested in the precise number of loop iterations, we are only interested in the maximum (and possibly minimum) power consumed by the loop. If this coarse-grained profile does not provide the desired level of detail, we further analyze inner loops during the next iteration. This way, we only perform the complex analysis of nested loop structures if the loop consumes significant energy. With *desired level of detail* we mean enough detail to show whether the key property is satisfied, as will be explained in the next subsection. The profiling step will not evaluate whether the acquired profile does have the desired level of detail. However, each iteration acquires a more detailed profile until the CEGAR loop terminates successfully or with a real counterexample.

After detecting that a given counterexample is spurious, it is easy to refine the model to one that excludes this counterexample. During refinement, however, the goal is to rule out a whole family of counterexamples, not just a single counterexample. Therefore, we have to generalize the model, which is where uncertainty comes in. For example, if the energy consumption of a loop is heavily dependent on the input data, it might only consume little energy for the specific input data used during profiling, but more energy for other input data. Thus, profiling must be performed with various input data until enough certainty is acquired on the resource consumption bounds.

Based on the counterexample, it is possible to automatically detect which ranges of instructions should be profiled. Once this is known, logging can be inserted for measuring entering and exiting these ranges. Since the model contains all bytecode instructions of the actual application, logging can be added by inserting the bytecode instructions for writing to the log in the model, after which the model itself can be executed. The test driver can steer the execution of the application by issuing input events. Which input events must be generated can also be derived from the counterexample.

When the application is being executed, the resource consumption must be measured with external tools. This can be done with software profilers, such as Trepn, or with hardware, for example provided by the SEFLab (the tools will be explained in section 5). These tools can measure the logged events, which is when a certain range is entered or exited, together with the energy consumption over time.

4.4 Refining abstraction

When a sequence has been profiled, the measured time and energy consumption can be generalized to energy bounds, which can be annotated on the sequence. This way, the model is refined with more precise timing and energy bounds, which can be used to validate whether the key property is satisfied or not. If the key property is indeed satisfied (as will be decided by step 2 of the CEGAR loop), the process finishes and the resulting model is the desired one.

If the key property is not satisfied, there are two possibilities: *(a)* the newly obtained profile violates the key property, or *(b)* the profiles do not provide enough information to show whether the key property holds on the actual application. In the former case, a concrete counterexample was provided (by step 3 of the CEGAR loop), which the developer can use to either correct the code or correct the key property, since one of these must be incorrect. In the latter case, we consider the counterexample spurious and the CEGAR loop continues with another iteration of profiling, during which more information will be acquired.

Due to the uncertainty introduced by profiling, the definition of *spurious* counterexamples is slightly different from the definition traditionally used by CEGAR. Traditionally, CEGAR considers a counterexample spurious if it *does not* occur in reality. Now, we consider it spurious if *we are confident that it will not* occur in reality. Therefore, the refinement process progresses in two directions:

- When we have enough confidence that the counterexample will never occur, we refine the abstract model such that the counterexample cannot occur anymore (as we already did in section 2.2).
- When we only have little confidence that the counterexample will never occur, we refine the profile with additional runs such that the confidence increases or the counterexample is shown to be real.

The last direction is newly introduced because we add profiling. For now, we decide whether we have enough confidence manually.

5. TOOLS FOR PROFILING

During profiling, the application is analyzed with external tools while it is running. We have evaluated several such tools and present the tools we consider most promising in this section.

5.1 Trepn

Trepn Profiler[2] is an energy profiler developed by Qualcomm. Trepn runs on most Android devices, but the accuracy of power measurements might differ depending on the device, and some features are only available on devices using a Qualcomm Snapdragon processor. Trepn can display data in real-time or store it in a log file for offline analysis [1]. It also offers detailed graphs that can help in analyzing how applications use energy. Trepn can analyze one particular application, or the device as a whole. Qualcomm also offers an Eclipse plug-in for Trepn, that developers can use to correlate battery spikes with system events. Using this plug-in, developers can import the data Trepn generates directly into Eclipse, which can make development more efficient.

[2]See the Quallcomm Developer Network website at `developer.qualcomm.com`.

5.2 JouleUnit

JouleUnit [6] is a framework for profiling the energy consumption of Android applications in order to generate energy labels. To provide such labels, the framework supports writing system tests that execute a scenario and measure the power consumption of the full device. Such measurements are performed by special hardware—e.g.: an oscilloscope—after which the framework synchronizes the measured samples with events generated through JouleUnit API calls. Such special hardware is required because energy measurements made by the smart phone itself are too coarse grained: generally, software generates events at a higher frequency than the internal battery sensor measures. The special hardware provides measurements that are fine grained enough to synchronize the measured samples with such events.

5.3 SEFLab

The Software Energy Footprint Lab (SEFLab) [2] executes software products on a server (or desktop system) and outputs power measurements taken during the software execution. This server is rigged with sensor boards attached to the power distribution lines that go from the power supply unit to the motherboard. This way, accurate independent power measurements are obtained for several components. The measurements are processed on a separate PC, which allows real-time visualization of the data as well as writing the data to file.

The main distinction between the SEFLab and software profilers such as Trepn is that the SEFLab has extensively validated the accuracy of their measurements, whereas software profilers sometimes provide very low accuracy (the actual accuracy is often unknown). Software profilers can provide valuable information for comparative analysis over a period of multiple seconds, but are insufficient for quantitative analysis or short events [3]. Also, of course, the platform that is used for Trepn—Android—differs from that of the SEFLab—server and desktop systems. The kind of data that is acquired from both approaches is similar, so the choice for a specific profiler is not inherent to our method and, therefore, does not limit the generality of our results. For its simplicity and because we perform comparative analysis only, we chose Trepn to experiment with. However, the high accuracy guaranteed by the SEFLab would make the SEFLab our preferred choice for future research.

6. RESULTS

We have applied our approach to a Java class of 72 lines of code, which periodically downloads a file from the Internet. When we extract a jbcpp model from this program, we get a rather large model, containing two classes and four methods. However, by applying our two transformation steps to *(a)* group sequences of nodes, and *(b)* remove unconnected nodes we acquire a much smaller model. This model still contains two classes and four methods, but the number of nodes inside the methods is reduced.

We translate this model to UPPAAL to perform our model checks. The UPPAAL model consists of three templates: one for each class and an additional (very small) one that drives the execution.

Model checking provides us with a trace, from which we can acquire the Trepn profile shown in figure 6. The left quarter of the graph differs from the rest, because it includes startup time and, therefore, should be ignored. The rest of the graph shows a repeating pattern, since our trace contains a loop. When we look at the bottom graph, we see the application states; the application switches between two states. In the graph above that, we see that one of these states corresponds to sending data through the WiFi network. This is also the state that consumes most energy, as shown by the graph at the top.

This profile provides enough information to update our jbcpp model with timing and energy constraints, such that the key property is satisfied. By applying our toolchain to this Java class, we show that we can augment extracted models that do not contain energy information with resource information acquired by profiling. Applying our approach on larger software application remains future work. However, we have already shown that interpreting energy profiles with CEGAR is a promising approach to automatically extract resource-utilization models from existing software.

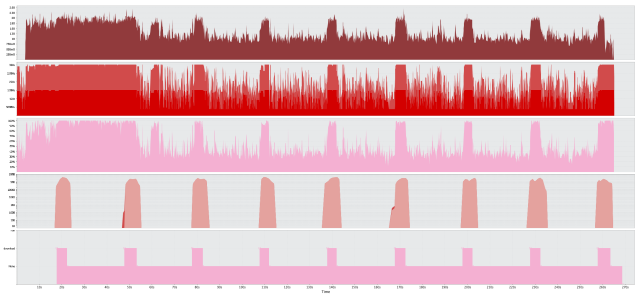


Figure 6: Trepn profile (the graphs represent: battery power, cpu frequency, cpu load, WiFi traffic, application states)

7. REFERENCES

[1] A. Bakker. Comparing energy profilers for Android. In *21st Twente Student Conf. on IT (TSConIT)*, Enschede, The Netherlands, June 2014. Univ. Twente.

[2] M. A. Ferreira, E. Hoekstra, B. Merkus, B. Visser, and J. Visser. SEFLab: A lab for measuring software energy footprints. In *2nd Int. Workshop on Green and Sustainable Softw. (GREENS)*, pages 30–37, May 2013.

[3] H. Höpfner, M. Schirmer, and C. Bunse. On measuring smartphones' software energy requirements. In *Proc. 7th Int. Conf. on Softw. Paradigm Trends (ICSOFT 2012)*, pages 165–171. SciTePress, July 2012.

[4] S. te Brinke, S. Malakuti, C. M. Bockisch, L. M. J. Bergmans, and M. Akşit. A design method for modular energy-aware software. Technical Report TR-CTIT-12-28, Centre for Telematics and Inf. Tech., Univ. Twente, Enschede, The Netherlands, Nov. 2012.

[5] S. te Brinke, S. Malakuti, C. M. Bockisch, L. M. J. Bergmans, M. Akşit, and S. Katz. A tool-supported approach for modular design of energy-aware software. In *Proc. 29th Annual ACM Sympos. on Appl. Comput. (SAC 2014)*, pages 1206–1212. ACM, Mar. 2014.

[6] C. Wilke, S. Götz, and S. Richly. JouleUnit: a generic framework for software energy profiling and testing. In *Proc. 2013 Workshop on Green in/by Softw. Eng. (GIBSE '13)*, pages 9–14. ACM, Mar. 2013.