Greening Software with Continuous Energy Efficiency Measurement

Jakob Drangmeister, Eva Kern, Markus Dick, Stefan Naumann, Gisela Sparmann, Achim Guldner

Environmental Campus Birkenfeld
Institut for Software Systems
P.O. Box 1380
D-55761 Birkenfeld
{j.drangmeister,markusadick}@gmail.com
{e.kern,s.naumann,g.sparmann,a.guldner}@umwelt-campus.de

Abstract: The energy consumption of information and communication technology (ICT) is still increasing. Several solutions regarding the hardware side of Green IT exist, until now the software contribution to Green IT is not considered sufficiently apart from scientific research. In our paper, we discuss a new method of improving the energy efficiency of software during its development, by putting energy efficiency measurements into practice. Therefore, we measure and rate energy consumption and efficiency during the software development process, based upon software testing and Continuous Integration (CI).

1 Introduction

With the increasing wealth and the growing global population, the efficient use of natural resources is becoming increasingly important, due to its limited extent. If humanity continues the excessive exploitation of planet Earth, it can be assumed that we are running out of natural resources in the near future [MMR06]. According to the estimates of the United Nations, the world population will increase to over nine billion people by the year 2050\(^1\). The gain of wealth outside industrialized countries is giving rise to an even stronger demand for resources, including ICT. While it is significant to produce hardware that consumes as little energy as possible during its lifetime, the resource-intensive production of modern ICT systems makes it even more important to use hardware components as long as possible.

Despite the fact that software plays an important role to hardware requirements and energy consumption during the usage phase [Gr05, CFS12, Di11, We12, Wi12], apart from scientific research and embedded systems, it is not common today to take the energy consumption and efficiency during the development of desktop and server applications into account. To solve this problem, it is necessary to raise the awareness of

developers management and clients towards efficiency and energy consumption induced by software. Above that, it is important to set a low entry barrier for measurement and rating energy efficiency while designing and writing software [BM09, Kr12].

Facing these challenges, we describe a method how to measure and rate the energy efficiency on an ongoing basis during the development process based on CI [Be99, Fo06]. Together with the growing popularity of agile methods since the beginning of the 90s, it became obvious that CI is an essential approach to optimize teamwork during software development. Today the use of CI is widespread and well-proven. In order to limit the additional effort for developers, our method integrates seamlessly into processes and tools that are already established in practice.

2 Related Work

To be able to measure and rate the energy efficiency of software, it is necessary to apply metrics and measurement methods. There already exist procedures and recommendations on both topics.

Bozzelli et al. [BGL12] provide an overview on green software metrics and conclude that there is not only one metric, but that a suitable metric depends on the domain the software operates in and the aim of the analysis. Johann et al. [Jo12] reach a similar conclusion. They suggest understanding energy efficiency of software as the ratio between useful work done and the energy the software consumes to do the useful work. Here, the definition of useful work done depends on the software type. Above that, they introduce a measurement method by instrumenting the source code to count the useful work while measuring the energy consumption with a power meter.

Weaver et al. [We12] provide a ready to use API to monitor energy consumption. To determine the energy consumption, they support power meters, as well as the values provided by Hardware Performance Counters (HPC) of modern CPUs and GPUs. Schubert et al. [Sc12] developed a software energy profiler, which enables the estimation of energy consumption of single functions or single source code statements, without the need of manual source code instrumentation or using a power meter.

Hindle [Hi12] measured the power consumption of a workload on many different versions of the Mozilla Firefox web browser. He showed that power consumption potentially varies between versions and comes to the result that it might be necessary to monitor the power consumption for each revision during the development process. Wilke [Wi12] used an application independent service model and a modified Android JUnit test runner to generate application specific workloads for measuring and rating the energy consumption of mobile applications. In addition, he gives an overall comparative rating to different applications of the same domains.

Common element of the described methods is that they all need exact knowledge of the implementation. Thus, they can be called white box measurement methods. These are
necessary to analyse and rate the energy efficiency of single modules or functions of the implementation.

Dick et al. [Di11] set a different focus. They implement black box measurements by executing workloads on deployed software products. Their aim was to compare the overall energy consumption of desktop and server products in the same domains.

3 Energy Efficiency Measurement with Continuous Integration

In this paper, we describe an approach that makes it possible to perform black box measurements, as well as white box measurements. The aim is to rate and improve the energy efficiency of single modules or methods during the development phase of a single software development project, to build a product as energy efficient as possible. To generate the workloads, we use established test procedures, for example unit and integration tests. The CI environment (Figure 1) is used to automate the test execution after each integration and to generate the energy efficiency reports. The energy consumption is monitored during the test execution with a hardware power meter and estimated with a simple CPU usage based power model.

![Diagram of Measure Energy Efficiency with Continuous Integration](image)

Figure 1: Measure Energy Efficiency with Continuous Integration
By automating the measurements and the creation of reports, the feedback to the developers is maximized and they acquire the ability making adequate choices on optimizations concerning minimizing the energy consumption and maximizing the useful work.

3.1 Using Software Tests to Rate Energy Efficiency

As mentioned before, we use well-known test techniques to generate the workloads in order to guarantee an integration of the measurements into the development process. Thus, we analyzed the terms and definitions on software testing and reviewed their suitability for rating the energy efficiency. We address the following requirements to a test:

- The test should generate a significant load over at least a few seconds, or better, minutes to measure the energy consumption during the test execution.
- It has to be possible to define a suitable metric, in order to rate the energy efficiency of a test execution.

The execution time of a few seconds is a concession to the resolution of the power meter and the necessity to capture significant values on energy consumption and the system load. The metric defines the useful work done during test execution. Because the metric should fit the aim of the analysis, it is necessary to define one metric per test.

Unit integration and system tests are regularly used to ensure the compliance of the software with the functional requirements. Because of the functional bias, one should always be able to address the useful work done during test execution. Unfortunately, these tests are usually designed to execute as fast as possible, which makes them inadequate for measuring energy consumption. Especially unit tests are mostly executed in a few milliseconds. Performance and load tests focus on non-functional requirements and provoke long execution times, for example to analyse the mean time on response. However, it is difficult to define function oriented metrics for non-function orientated tests. Thus, we recommend combining both aspects and writing performance and load tests that focus on functional requirements. For example, these tests could be based upon already implemented unit tests. The unit tests could be executed several times or compute larger problems.

3.2 Monitoring Test Execution

Most projects use build environments, to automate the whole build process. A build environment compiles and distributes the software. Above that, it handles the test execution during or after distribution by means of a test framework. The CI environment initiates the build. After completion, the build environment notifies the CI environment about build and test results.
To monitor the energy consumption, the useful work done, and the system load, the test execution has to be extended. The measurement should start and end with a test, the measurement results should be added to the test result and saved to the output files together with the test result (Figure 2).

We decided to record the following values during test execution:

- **Total Energy Consumption**: The electrical work needed to execute a test. It is measured by the power meter.
- **System Load**: If a power meter is not available, we estimate the energy consumption based on test execution time and the system load.
- **Metric Name**: A suitable description to specify the useful work done during test execution and to rate the energy efficiency [Jo12].
- **Metric Counter**: The counter for the metric to capture the useful work done during test execution and to rate the energy efficiency [Jo12].

### 3.3 Reporting Energy Efficiency

The Energy Efficiency Report generated by the CI environment enables developers to get a quick overview on the chronological progress of energy efficiency of the whole project during its development. If changes occur, it is possible to get detailed reports for individual test packages, classes and methods. This is necessary for developers to identify the source of changes. The report contains the measurements made during test execution and shows a rating of the energy efficiency towards [Jo12]:

\[
\text{EnergyEfficiency} = \frac{\text{UsefulWorkDone}}{\text{UsedEnergy}}
\]
Prototypical Application of the Method

To evaluate our method, we needed an application to simulate a change in energy consumption during the development process. We wrote a small Java project called Sorter implementing the sort algorithms heapsort and quicksort. Knowing that these algorithms differ in complexity (Table 1), we were confident to provoke a variation in energy consumption by changing the sorting algorithm used by Sorter.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst case</th>
<th>Average case</th>
<th>Best case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$</td>
<td>$O(n \cdot \log(n))$</td>
<td>$O(n \cdot \log(n))$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \cdot \log(n))$</td>
<td>$O(n \cdot \log(n))$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Table 1: Complexity of Quick sort and Heapsort

Sorter takes an array of unsorted integer values, sorts the values and returns the sorted array. We wrote several unit tests to ensure that the algorithms work properly. Above that, we implemented three performance tests, based upon the unit tests, to rate the energy efficiency of both implementations:

- SorterUnsortedTest: Sorts an array with 10 million unsorted integer values and compares the resulting array with the expected result.
- SorterSortedTest: Sorts an array with 10 million already sorted integer values and compares the resulting array with the expected result.
- SorterSameTest: Sorts an array with 10 million equal integer values and compares the resulting array with the expected result.

In these cases, defining a suitable metric is simple. Obviously, the useful work could be addressed as sorting integer items, so the metric is:

$$\frac{Sorted\text{Integer}\text{Items}}{Used\text{Energy}}$$

The experimental set-up to perform the application depends on a set of components. We used Jenkins\(^2\) as CI environment. The build process of the Java project was automated with Apache Ant\(^3\). The tests were executed by TestNG\(^4\). The processing of the TestNG

\(^2\) http://www.jenkins-ci.org, accessed 11 Apr 2013
\(^3\) http://ant.apache.org, accessed 11 Apr 2013
\(^4\) http://www.testng.org, accessed 11 Apr 2013
report was executed by the TestNG-plugin\(^5\) for Jenkins. The system load was gathered with SIGAR\(^6\) and the energy consumption was recorded with a power meter.

The test classes contain three noteworthy methods respectively. The method `create()`, marked with annotation `@BeforeClass`, is executed once when the test class instance is being created. It imports two predefined arrays of integer values from text files, one with the expected result and one in randomized order that is to be sorted during the test run. The method `createTest()` (Figure 3) is annotated with `@BeforeMethod`. Hence, it is executed before each test run and used to get and extend the TestNG test result with the name of the test metric via the interface `ITestResult`, offered by TestNG.

```java
@BeforeMethod
public void createTest(ITestResult tr) {
    tr = tr;
    tr.setAttribute("usefulWorkDoneName", "SortedIntegerItems");
}
```

Figure 3: The Method `createTest()`

The test method (Figure 4) adds the useful work done attribute value to the test result and calls the `sort()` method of the Sorter class. The `sort()` method sorts the contents of the array and returns it. Finally, the contents of the sorted array are compared to the contents of the array containing the expected result.

```java
@Test(threadPoolSize = 1, invocationCount = 30)
public void sortUnsortedList() {
    testResult.setAttribute("usefulWorkDoneCount", unsortedList.length);
    Assert.assertTrue(Arrays.equals(sortedList, Sorter.sort(unsortedList)));
}
```

Figure 4: The Method `sortUnsortedTest()`

The `@Test` annotation option `invocationCount` is used to execute the test method 30 times for being able to minimize the error of measurement by calculating average values in the energy efficiency report (section 4.2).

### 4.1 Monitoring TestNG and Extending Test Results

To extend the test results with the measurements, we implemented three components and embedded them into the test execution. The `PerformanceMonitor` is used to gather the average CPU load, the `EnergyMonitor` polls the energy consumption over the testing period.

TestNG offers a map of attributes with each test result to add individual values. To add the measurement values to the test results, we implemented the interface `ITestListener` as `GreenTestListener`. The `GreenTestListener` observes the start and end events on test

\(^6\) http://www.hyperic.com/products/sigar, accessed 11 Apr 2013
execution and calls methods to add the attributes with the measurements delivered by the Performance and EnergyMonitor (Figure 5).

```java
private static void initOnTestStart() {
    enerMon.startCounters();
    perfMon.startTickler();
}

private static void setAttributesOnTestFinished(ITestResult result) {
    result.setAttribute("usedEnergy", enerMon.getUsedEnergy());
    result.setAttribute("averageCpuUsage", perfMon.getCpuUsage());
    perfMon.stopTickler();
}
```

Figure 5: Methods used to add Measurements to Test Results

The measurement data added during test execution is stored together with the test results to the TestNG XML report for subsequent processing in the CI environment.

4.2 The Energy Efficiency Report

The Energy Efficiency Report was integrated into the Jenkins web interface by forking the source code of the TestNG-plugin and extending it with the handling of the measurement data. Following our recommendations in section 3.3, we implemented a project overview (Figure 6) and the detailed report (Figure 7) to show the energy efficiency of test packages, test classes, and single test methods.

Figure 6: Project Energy Efficiency Overview
The project overview report shows the trend of the average energy efficiency of all test methods executed during a build. Within the report, it is possible to open a single build and browse through the packages, classes, and methods. Those are displayed with the detailed report. If a package or a class is displayed, which contains more than one test method, the report shows the total, average, and standard deviation for the measured values. Above that, it shows the energy efficiency trend of the currently selected method, class, or package. Figure 6 displays the overview report with the projects energy efficiency trend. In Figure 7, the detailed report is shown, generated for test class SorterUnsortedTest of build #37. As mentioned in section 4, each test was executed 30 times in series, the total column contains the duration of all executed tests while the average column displays the arithmetic mean.

Figure 7: Detailed Energy Efficiency Report

4.3 Measurement Example

Our measurement example was executed on a desktop computer (Intel Core i3 540 @3,07GHz, 4GB RAM, 36GB WD360). We measured the idle consumption with 46 Watts, the consumption on maximum load was determined to 89 Watts. We decided to use the Ubuntu GNU/Linux (12.04 64Bit, Kernel 3.2.0-35-generic) operating system. To compile and execute the Java project, we installed the Oracle Java Development Kit 1.6.0_38. In a real life scenario, the build and CI environments should be executed on separate machines to avoid influences on measurements caused by activity on the CI server. Our setup was simplified by using this machine as test execution environment and CI environment at the same time. We executed only one build at once and we ensured that there is no activity at the CI environment during a build. Thus, we were able to consolidate them.
As described before, we implemented the Sorter application to simulate a significant change caused by a switch between the used sorting algorithms. The assumption was that with the extended test execution and the Energy Efficiency Report, it should be possible to recognise a significant change in energy consumption and energy efficiency. To verify this, we activated quicksort and uploaded the source code into the version control system. Hence, build #37 contains a rating of our heapsort implementation. After finishing the test execution, we activated quicksort, so build #38 contains a rating of our quicksort implementation. In order to take errors in our measurements into account, each test was configured to execute 30 times. Jenkins automatically executed and saved a build for both cases containing test and average (arithmetic mean) measurement results (Table 2).

<table>
<thead>
<tr>
<th></th>
<th>Build #37 (Heapsort)</th>
<th>Build #38 (Quicksort)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SorterUnsortedTest</td>
<td>24.687 Joule</td>
<td>97.087 Joule</td>
</tr>
<tr>
<td>SorterSortedTest</td>
<td>69.268 Joule</td>
<td>327.868 Joule</td>
</tr>
<tr>
<td>SorterSameTest</td>
<td>N/A</td>
<td>113.079 Joule</td>
</tr>
</tbody>
</table>

Table 2: Results of the Measurement Example

Hence, with the described test cases our implementation of quicksort has, as expected, better energy efficiency as our heapsort implementation. The results generally match the expectations. Although quicksort has the higher worst case and both algorithms have the same average case complexity, quicksort is typically 2-3 times faster than heapsort [Sk08]. The measurement for SorterSameTest on heapsort could not be applied because heapsort did the work in less than one second, so underruns the measuring resolution of the power meter. The low run time may be explained by the fact that SorterSameTest represents the best case O(n) of heapsort.

Thus, with SorterUnsortedTest and SorterSortedTest we showed that it is possible to recognise the expected difference using our method and quicksort is the better choice for our example project.

5 Summary and Outlook

With the rating of sorting algorithms, a rather simple example was chosen. It is easy to define a metric to rate sorting algorithms, while it could be difficult for software developers to define suitable metrics on complex applications and white box tests. Therefore, it is important to take the tests for rating the energy efficiency into account right from writing the first line of code. Testing code is easier when the code is prepared for testing, so it may be useful to write the tests and the code test-driven. The additional expenses incurred by testing the energy efficiency are potentially low when software testing is well-known and widespread in a software development project. They are high when tests are solely written to rate energy efficiency. Our example shows the application of white box tests, because we use standard approaches coming from...
standard software testing. But it should be possible to adapt our method for executing black box tests as well, e.g. for rating the energy efficiency with integration and system tests.

The capture of the useful work done during test execution could be improved. In our example we add the counter to the test result by knowing the problem size is computed during test execution. With the exception of whether the test succeeds or fails, we are not able to ensure the test actually does what is expected. This data can only be captured by instrumenting the source code manually or by the help of profiling tools. Additionally, it could be relevant to further split the metrics, e.g. for sorting algorithms to count comparisons or swaps.

It is difficult to capture the energy consumption with a power meter on build and CI environments that manage more than one project, e.g. which are virtualized or placed in data centres. The problem is that there will be the possibility that more than one build or test is executed at the same time or another virtual machine causes energy consumption while executing the tests. These problems will frequently occur in practise and can only be solved by capturing energy consumption in a different way. A suitable approach could be to estimate the energy consumption by system load or HPC caused by a single test and its execution time.

The measurement method described in this paper helps to monitor and optimize the energy efficiency as early as possible: during the software development process. In summary, the method puts energy efficiency measurement of software into practice. With our general focus on rating any kind of software with software testing, developers are enabled to inspect server and desktop software during its development, as well as mobile applications. It is suitable to rate a whole product, as well as single modules or methods and monitor their evolution regarding energy consumption and efficiency from the beginning to the end of a software development project. It can be easily integrated into the daily work of software developers, without breaking their usual working processes. Thus, it could be an important element to develop greener software in the future.

References


