Methodological Guidelines for Measuring Energy Consumption of Software Applications

Methodological Guidelines for Measuring Energy Consumption of Software Applications

Luca Ardito, Riccardo Coppola, Maurizio Morisio, Marco Torchiano

Politecnico di Torino
Department of Control and Computer Engineering
Turin, Italy
name.surname@polito.it

Abstract

Energy consumption information for devices, as available in the literature is typically obtained with ad-hoc approaches, thus making replication and consumption data comparison difficult. We propose a process for measuring the energy consumption of a software application. The process contains four phases, each providing a structured deliverable that reports the information required to replicate the measurement. The process also guides the researcher on a threat to validity analysis to be included in each deliverable. This analysis ensures better reliability, trust, and confidence to reuse the collected consumption data. Such a process produces a structured consumption data for any kind of electronic device (IoT devices, mobile phones, personal computers, servers, etc.), which can be published and shared with other researchers fostering comparison or further investigations. A real case example demonstrates how to apply the process and how to create the required deliverables.

Keywords: energy consumption, software energy consumption, software application, software engineering, energy consumption data, computer engineering

1. Introduction

A software program contains a sequence of instructions whose execution requires the device on which it is running to consume energy. Today, energy consumption, a non-functional property of the program, is seldom considered upfront as a non-functional requirement or, after the fact, as a property to

be measured and monitored. However, energy consumption may represent a critical problem for end users. In laptops, tablets, and smartphones, energy consumption clearly has an impact on battery life and, therefore, it becomes a user experience issue [1]. For data centres or Bitcoin miners [2], energy consumption has a direct impact on the electrical bill. In the literature, many have addressed the problem of measuring and reducing energy consumption, but typically in an ad-hoc manner [3].

According to the Evidence-Based Software Engineering (EBSE) [4] approach, concrete decision-making should be supported by the empirical evidence available in the literature. Such evidence must be trustable, produced through a documented and repeatable process, contextualised, linked to the context where it can be applied, identifiable, address a well-defined question, assessable, and report the known limitations of the results. Such characteristics are seldom present in most of the related published literature.

If the energy consumption issue is tackled at the hardware level, then the task is accomplished by reducing the consumption of the physical devices or by creating different usage profiles (e.g., processors can scale down the frequency when used less). On the other hand, if the energy consumption issue is managed at the operating system level, then management policies may use the different hardware profiles of various devices (when available) or turn off hardware when not needed. We consider software as a driver of the energy consumption because it requires several actions to be completed by the underlying hardware, which reacts based on the received instructions. Measuring the energy consumption due to a specific piece of software implies addressing two major issues:

- Isolating the energy consumption of a program when it is running concurrently to others on the same device.
- Generalizing the obtained results: let the measure be meaningful to other devices.

When collecting energy through physical measurements on a device, the value is related to the target software and all other processes running on the device simultaneously. The physical measurement does not allow a straightforward generalisation of results because the same software could behave differently based on the hardware on which it is executed as well as other installed software. Another option is defining models that provide an estimate of the energy consumption of the target software instead of performing a

physical measurement. The input of the model consists of device resource usage indicators collected at run-time. The main issue affecting this approach is that a model can be representative of a device or a family of devices meaning that the estimation computed by the model is not always valid. Unfortunately, it is very difficult to get this result because every hardware manufacturer should provide accurate data on the consumption of the device, and this data should be available in real time as device status information through sensors and system calls from the operating system. Now, we can easily measure the energy consumption of an application by measuring the energy consumption of the entire device on which it is executed, analysing the obtained data, and estimate the consumption by minimising the error. This requires a precise methodology to obtain the most significant data and analyse them for useful information to estimate the power consumption of an application.

In this paper, we propose a general process that can be used to measure the energy consumption of a software application. This process includes the best practices for collecting and analysing energy consumption data of a software application and formalises the steps needed to carry out a valid empirical experiment. Thus, this is proposed as a ground zero for performing software energy measurements to ensure repeatability and comparison of each experiment. The process we put forward can be used both to conduct energy measurement and to asses existing studies serving as a sort of checklist.

The remainder of this paper is organised as follows:

- Section 2 describes the proposed process to collect energy consumption data from devices as well as how to analyse the data;
- Section 3 provides a real case study showing how to create the deliverables;
- Section 4 reports the related work and assess the literature in terms of compliance with our process;
- Section 5 concludes the manuscript and provides hints for future work.

2. Software Consumption Measurement Process

The proposal described in this paper is a repeatable process for measuring the consumption of a software application, hereinafter called the Software Under Test (SWUT). The process consists of the following four phases :

Table 1: Deliverables of the different phases and impact on threats to validity

Phase	Input	Output	Threats to Validity
Goal	-	SWUT and Context Research Questions Devices	External : Generalization of results
How	Goal Deliverable	Instrumentation	Internal : Assigning consumption value to a process
		Synchronization	Construct : Incorrect Measurement
		Sampling	Conclusion : Insufficient number of repetitions
		File Format	
Do	How Deliverable	Measurement Scripts Data Files	Construct : Incorrect Implementation
Analyse	Do Deliverable	Data Analysis Scripts Results and Discussion	Conclusion : Not suitable statistical tests

- Goal (G): define the research question, the target device(s) on which the measurements will take place, and the context in which the SWUT is executed.
- How (H): decide how consumption will be measured and the procedure needed to carry out the measurement.
- Do (D): carry out the measurement and collect the data.
- Analyse (A): analyse the data and address the research question(s).

The UML Activity Diagram in Figure 1 summarizes the main activities and decisions encompassed by the process and the relative threats to the validity of the results.

Each phase of the process shall produce a deliverable, which summarises the decisions taken, the outcomes of the phase, and the said analysis of the threats to validity. A summary of the elements provided by each deliverable is provided in Table 1. As it is evident in the table, each deliverable serves as an input for the following one.

The following subsections describe each phase of the process along with the required information to reproduce it.

A sample application of the described process to a simple case study will be then described in Section 3.

Each phase requires a few decisions to be taken, some of which can influence the validity of the results. Wholin et al. [5] classified the threats to

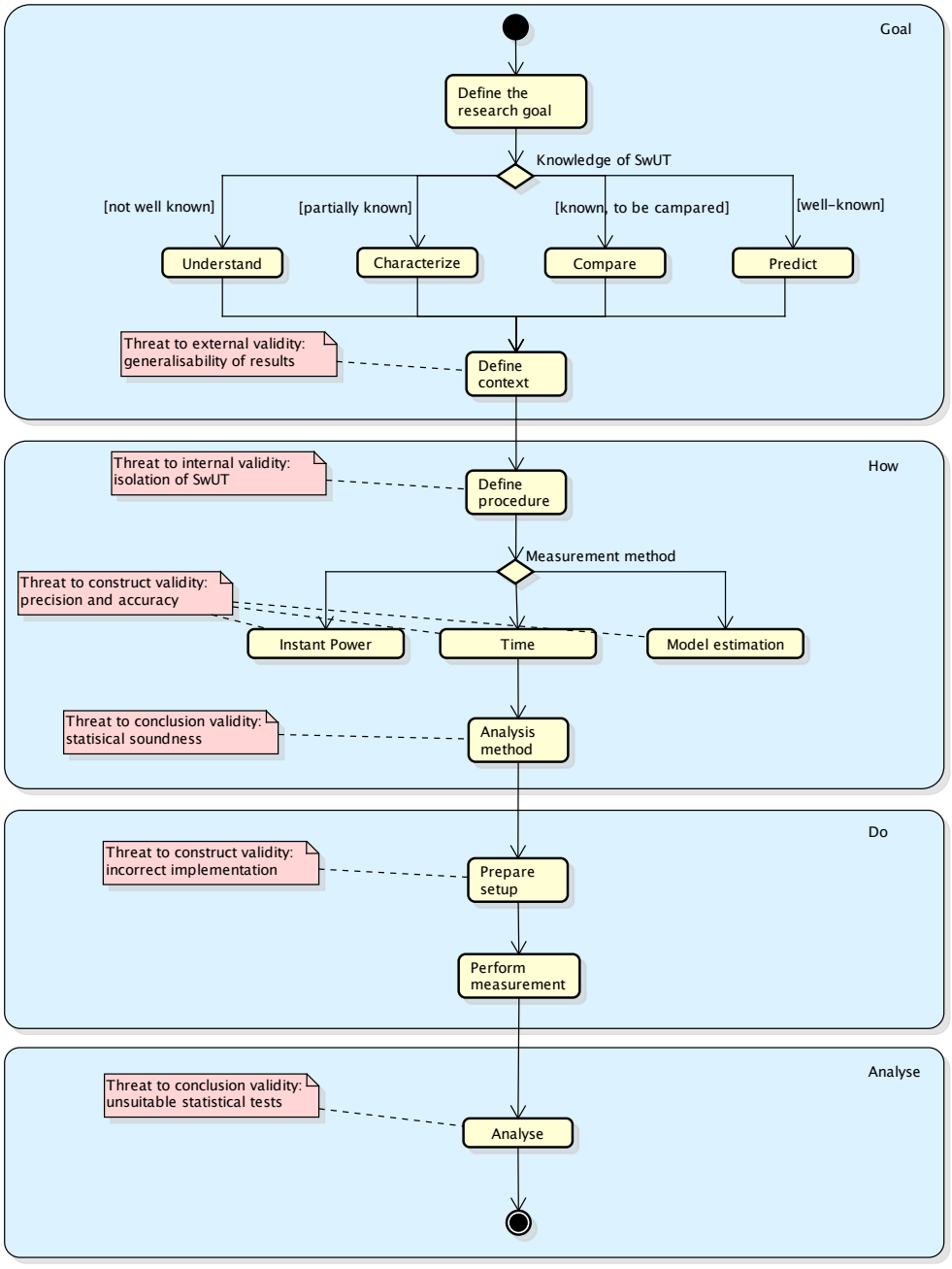

Figure 1: Summary of activities and decisions of the proposed process.

validity as:

- Internal Validity: focused on how sure it is that the treatment actually caused the measured outcome;
- Construct Validity: focused on the relation between the theory behind the experiment and the observation;
- External Validity: focused on the generalizability of the results outside the scope of the study;
- Conclusion Validity: focused on the relationship between the treatment used in the experiment and the actual outcome measured.

Table 1 shows these categories of threats and how they are impacted by each phase of the process.

2.1. Phase I: Goal

This phase is about defining the research questions that will drive the measurement process. Since the scope of the research questions is restricted to energy consumption, we propose to represent the goal as a template inspired by the GQM approach [6]:

> "< understand | characterize | compare | predict > the <consumption> of the <SWUT> run on <device(s)> in <context(s)>".

An example of a research question obtained applying this template is:

> *"Characterize the energy consumption of the Bubble Sort algorithm implemented in Java language when run on Raspberry Pi version 2B in the context of Raspbian Linux OS".*

The first aspect to consider is the purpose of the measurement, which depends on the level of knowledge of a specific process, and includes the following options:

- Understand: this goal applies to the initial investigations for a process that is not well known to understand the input and output variables of the process. Nominal or ordinal measures may characterize variables.

- Characterize: this goal applies to a process that is partially known to enhance its description by providing the input, output, and context variables that influence the process. Interval, ratio or absolute measures may characterize variables. Relationships between the variables, either analytic or probabilistic, are proposed, but their validity is limited.

- Compare: this goal is a variant of characterising where two similar SWUTs are characterised and compared on the variables defined.

- Predict: this goal applies to well-known processes to provide a model that relates all variables in the process. The validity of the model is broad, so that output variables are predicted by input variables reliably.

Consumption can be measured in terms of energy *[Joule]* or power *[Watt]*, which are related and one may be computed from the other. However, in practice they are not entirely interchangeable:

- From the research goal, power offers an immediate view and is suitable for tasks with a very long (possibly infinite) duration, while energy is suitable for tasks with a finite duration. For instance, if the software function to be measured is "read an email message", or "convert an audio file from mp3 to wma", then energy is the most suitable to characterise the consumption of the functions. If the software function is "control the speed of an engine", then power is the most suitable.

- From the measurement as a function of the hardware configuration (server vs desktop vs mobile phone), it may be way easier to measure power compared to energy, which will be discussed in Section 2.2.

The SWUT can represent a function, a set of functions, a software process, a software application or a software application subset of features. The description of the SWUT includes the programming language, the toolchain used to produce it (e.g., the compiler and its version or the linker and its version), and the usage scenario. Harman and colleagues [7] identified three levels of SWUT granularity: fine grained, corresponding to individual lines of code or statements; mid-grained, that is a block of code or a method/procedure; or coarse-grained addressing a whole program execution over a period of time.

The device represents the physical device (or devices) specifications (make, model, version, CPU, architecture, and memory) used in the experiment.

The context describes other attributes that may influence the experiment, such as:

- The operating system;
- The list of processes running while the measurement is performed;
- The device configuration;
- Any hardware and software instrumentation used to collect the energy information.

Since a SWUT can be very complex, addressing the research questions may require the creation of many subgoals, which aim at measuring the energy (or power) consumption of a predefined subset of features of the complex SWUT. We will provide a complete example in Section 3.

As seen in Table 1, the decisions must consider the threats to external validity, which regard the generalisation of results:

1. Threats help identify wether the results are valid only for the analysed device(s) and context(s) or they have wider validity.
2. Threats define the importance the obtained results will be valid on other devices or contexts. If yes, then researchers should state how device(s) and context(s) should be selected to minimise the external threats to validity. If not, researchers should state if it is in the goal of the experiment to obtain results only for a specific device and context.

This type of analysis during the early stages of the process has a twofold contribution. It makes the experiment more precise and formal as well as forcing who is experimenting to choose the best context(s) and device(s) to reach the goal.

The output of this phase is a deliverable which contains the goal description comprised of research question(s), device(s), SWUT, context, and the external threats to validity analysis.

2.2. Phase II: How

With the unit of measure (energy or power) determined in the first phase, this step will decide how to take the measurement. The three options are described in the following, whereas Table 3 analyses the benefits and drawbacks of each technique.

Instant Power Measurement. This technique measures the instantaneous current consumed by the device and then multiplies this value by the voltage. The integral over a period gives the energy value. Instant power measurements are precise if the sampling frequency is high, but they require physical instrumentation. This approach usually operates at the device level – although hardware component-level measurement is possible – and can work with coarse grained SWUT only.

Time Measurement. Another way to collect the energy consumption of a device is through measurement of time. A fully charged (and healthy) battery holds a known amount of energy (e.g., 1000 mAh corresponds to 18 kJoules). Assuming a constant consumption over time, the speed at which energy is depleted depends on the power consumption of the device. So, the average power consumed is computed by measuring the time to discharge the battery completely. This measurement relies on the precision of the battery capacity measure. If this value is imprecise, then so will be the calculated consumption value. Another issue is how linearly the battery discharges, especially if a measure is collected without fully discharging the battery. For devices without a battery (e.g., SoC computers, such as a Raspberry Pi), the type of measurement is possible by connecting the device to a battery instead of connecting it to the electrical network. This approach has the same limitation as the previous one in terms of granularity.

Model Estimation. Consumption measurements through models are calculated in a way that relates the power consumption of a particular device with internal resource usage indicators, such as the CPU states, instructions, memory or disk accesses, and network adapters. In the literature, there exist few examples of power models. For example, A. Patak et al. [8] described a power module based on system call tracing. This approach uses system calls for estimating the resource usage. Di Nucci et al. [9] proposed a software-based approach, named PETRA, proving that those methods are not inherently less precise than hardware-based or model-based solutions. Their approach is specifically aimed at testing Android applications. A. Nacci et

Table 2: Elements of the How deliverable

	Instant Power	Time Measurement	Model Estimation
Hardware Instrumentation	✓	✓	-
Software Instrumentation	-	✓	-
Synchronization	✓	-	✓
Sampling Frequency	✓	-	✓
File Format	✓	✓	✓
Threats to Validity	✓	✓	✓

al. [10] introduced an approach to build a power model for Android devices by using Android APIs to retrieve a variety of states, including the battery, network connection, Wi-Fi, and screen. Two components usually implement the models:

- A resource usage analyser that measures the usage of resources on a computer, which depends on the operating system;
- A resource usage to consumption converter that reads the data provided by the resource usage analyser and, based on the mathematical model, it converts to consumption values. The mathematical model is a parameter that varies according to the device.

The latter component requires choosing a model suitable for the device on which the SWUT will run. The model should provide the estimation error, the sampling frequency at which the resource usage is updated, and the overhead caused by extracting the resources utilisation. The overhead is a crucial value because a software process implementing the model executes the resource usage data collection, and, as with all the other software processes, it affects the consumption of the device on which it is executed. The sampling frequency and overhead are directly proportional.

This latter approach has the advantage of being applicable also to a fine grained SWUT.

The output of this phase is called *How Deliverable* as described in Table 1, which contains the key decisions used for obtaining the consumption of the SWUT. The deliverable will contain different elements based on the selected measurement approach, as shown in Table 2.

Table 3: Evaluation of measurement techniques

Measurement Technique	PROS		CONS	
	Energy	Power	Energy	Power
Instant Power Measurement	Precise if sampling frequency is high	-	Physical Instrumentation needed. Difficult to isolate a single software application's contribution.	
Time Measurement	Precise if the exact energy stored in the battery is known	-	Requires many repetitions of long tasks. Difficult to isolate a single software application's contribution	
Model Estimation	No instrumentation required. Easy to isolate a single software application's contribution		Precision not always declared	

The components of the How deliverable, and the way they vary according to the selected approach, are detailed in the following subsections.

2.2.1. Hardware Instrumentation

Hardware Instrumentation is required by the approaches based on Instant Power and Time Measurement.

Instant Power. To perform power measurement, the following hardware instrumentation is required:

- A Voltage Generator;
- A shunt resistor (e.g. 0,05 Ω);
- An ADC (analog-to-digital converter);
- A supervising device.

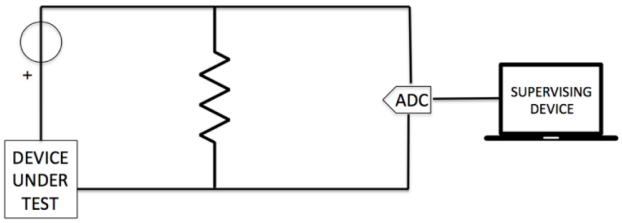

Figure 2: Circuit designed to measure instant power consumption

Figure 2 shows a typical configuration to measure instant power consumption data from the device. An ADC reads the voltage drop V across the shunt resistor. This data is sent to the supervising device, which will be later used for the analysis. According to Ohm's Law, V/R provides the current I, so the instant power consumption is calculated by $P = V \cdot I$. If the device has a battery pack, it should be removed because the voltage generator will also charge the battery pack during the experiment, providing inconsistent values to the ADC. Uncertainty on the power is $u(P) = P * (u(V)/V + u(I)/I)$. Both uncertainties are due to measurement errors and are typically relatively small when using suitable devices. On the market there are several power meters that can be used for the different categories of devices (e.g. mobile phones or single board computers, PCs, etc.). It is not required to build a power meter, however, its internal structure can be simplified to the circuit described in Figure 2.

Time Measurement. As described in Figure 3, a supervising device takes the system times during the test run, when the battery level changes, and when the device battery is completely discharged. For automating the time measurement, a programmable switch (represented by the dotted line connection between the supervising device and the switch) may be used to manage the charging process of the battery when it reaches a predefined discharge value (e.g., 2%). If the battery information is not available, then the predefined discharge value is 0%, and the device under test will turn off. Here, the problem is how to trigger this event. An example could be reading the output voltage value of a USB port with an ADC. When the voltage starts decreasing, the device is turning itself off, so this event can trigger the battery recharge.

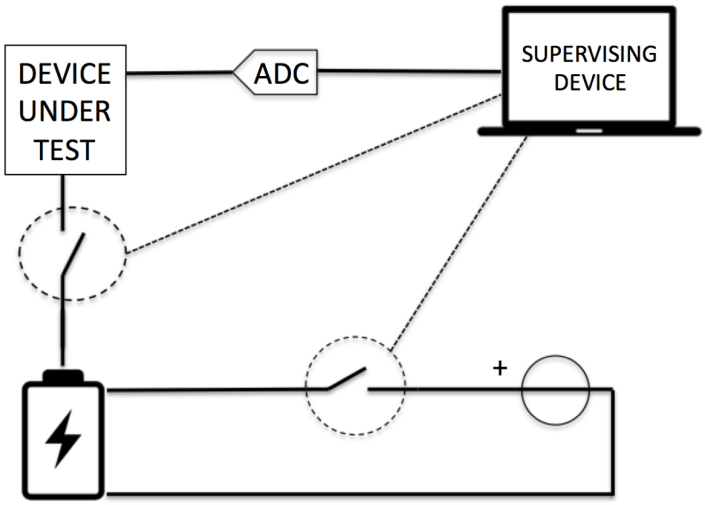

Figure 3: Hardware configuration for time measurement

2.2.2. Software Instrumentation

The Software Instrumentation is required only if the Time Measurement approach is selected. Time measurements require an automated procedure, which calls the SWUT continuously until the battery is discharged. At the end of the measurement, the result is an average consumption of the entire test run. To summarise, the measurement procedure should perform the following steps:

- Charge the battery until maximum battery level;
- Record the system time;
- Run the SWUT inside a loop until the battery is completely discharged. Typically, the SWUT is not able to completely discharge the battery in a single execution, so it must be run many times in an infinite loop while recording the number of runs;
- Record system time when the battery charge level changes (if this data is available);
- Re-record the system time when the battery charge level reaches a pre-defined minimum value or when it is completely discharged. Compute the experiment total time T and the number of runs, and then store the results in a file;

- Recharge the battery until it is fully charged;
- Repeat these steps to obtain reasonable statistics (e.g., 30 data points represents a meaningful dataset [11]).

Once the raw data is collected, the average power consumption is computed by analysing the time spent to completely discharge the battery as $\overline{P} = (C/T) \cdot V$, where:

- $\overline{P}$ is the average power consumption consumed in an hour,
- C is the total capacity of the battery in mAh[1],
- T is the time needed to discharge it in hours, and
- V is the voltage provided by the battery.

While the total energy consumed can be computed as $E = C \cdot V$.

The uncertainty on the $\overline{P}$ is $u(\overline{P}) = \overline{P} * (u(C)/C + u(T)/T + u(V)/V)$, thus depending on:

- $u(C)$: the uncertainty on the actual battery capacity, this is the most critical since battery tend to change their capacity over time and even new batteries might have actual capacity quite different from the nominal one;
- $u(T)$: the error in the time measurement: this error is typically small since complete battery discharge requires a long time;
- $u(V)$: the error in the voltage measurement: this error must be minimized using suitable measurement devices.

This technique assumes a constant power consumption value over the entire battery discharge time.

[1]Or the totale capacity minus the residual capacity at the predefined minimum.

2.2.3. Synchronization

Instant Power. In this approach, the consumption data – collected with a certain sampling frequency – is available on the supervising device used for collecting the data. However, power consumption must be associated with the process executing the SWUT, and this information is available on the DUT. In other words, it is needed to synchronise the time scales of the DUT and the supervising device. This problem can be solved in two ways:

- Synchronize the DUT and the supervising device system times so that each sample belongs to a known timestamp.
- Instrument the code by adding distinctive power patterns for a defined period before and after each run.

The first approach requires accurate time synchronisation between the DUT and the measurement device to record only the consumption related to the SWUT execution. The synchronisation could be achieved using NTP (Network Time Protocol). However, this solution can cause errors of more than 100 ms due to network congestion. It also requires both the supervising device and the device under test to be connected at least to a LAN to reach the NTP server. An error in the synchronisation between the two devices can lead to data invalidation, especially in experiments carried out in cascade because the consumption data collected is not entirely related to the SWUT. The second approach allows for the association of the consumption to a SWUT without synchronisation by adding markers in the SWUT. The markers are known as code patterns, which produce distinctive data consumption patterns identifiable by data analysis after the data collection, and may be defined as:

- Busy Marker: a function executing an empty infinite loop.
- Sleep Marker: a call to the sleep function.

It is possible to automatically identify these well-known patterns in the consumption data using signal processing techniques because the busy marker has very high power consumption, while the sleep marker has very low power consumption (see section 2.4). Figure 4 presents three busy markers, two sleep markers, and one execution of the SWUT tagged as work.

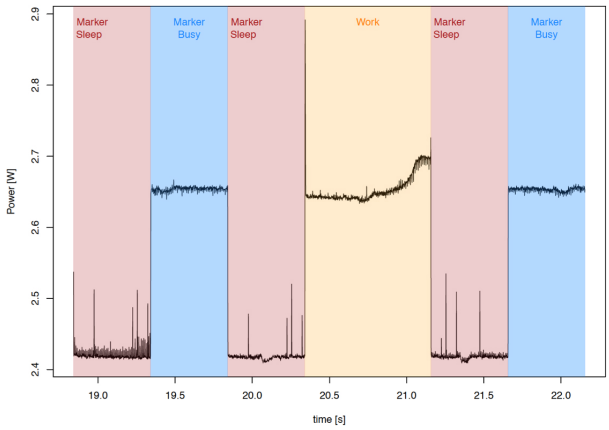

Figure 4: Consumption data and instrumented SWUT

Model Estimation. The problem related to time and data synchronisation is similar when this approach is adopted. Instead of having a consumption value, there will be resource usage data. It will then be required to translate the resource usage data into consumption data allowing the use of a timestamp to isolate the consumption data related to the SWUT. Alternatively, it is possible to add a marker before and after the SWUT to identify the SWUT consumption data between the two markers. While the latter approach can be followed exactly, the first approach is more straightforward because the model estimation does not require a supervising device, and hence there is no need to perform clock synchronisation. So it is possible to isolate the SWUT consumption data by using timestamps.

2.2.4. Sampling Frequency

A sampling frequency is required when the Instant Power and Model Estimation approaches are adopted. The instant power consumption measurement represents the average power consumption in each sample. A suitable sampling frequency is 125 kHz because only 1% of energy is consumed above this frequency as stated by Saborido et al. [12]. The authors stated that a 10 kHz measurement could lead to an error of 8%, so such a low sampling frequency causes significant errors.

The size of the data log should be considered as another constraint. Considering that each sample could be $\sim$ 10 bytes, at 10 kHz frequency the script produces $\sim$ 100 Kbytes per second. So, the sampling frequency should be selected carefully based on the duration of the process running the SWUT,

the related size of the data logged, and the acceptable error.

The same sampling frequency tradeoffs are valid for the Model Estimation approach. However, it should be taken into account that logging the resource usage too frequently can cause a sensible overhead.

2.2.5. File Format

The How Deliverable document contains the explanation of the raw data file format that is used.

For Instant Power measurements, the raw data is included in a plain text file with each line containing the instant current in A in the sample time T. This format is simple, easy to read, and does not contain any extraneous data. If the instant measurement contains multiple data (e.g., current, voltage, and the current system time), then it is better to organise the file in JSON or XML format to explicitly express the type of data included in the file. Such a definition of a file format and content is useful for creating data file parsers. The same file format can be applied to the Model Estimation approach, given that the output provided by the model is parsed and converted into consumption.

In case of Time Measurement, the raw data is included in a plain text file representing the duration of the experiment. This format is simple, easy to read, and does not contain any extraneous data, for example, 1:15:13.041454.

2.2.6. Analysis method

The typical goal of an energy measurement campaign is to assess whether any main factor, e.g., a specific algorithm or computation architecture, affects the energy consumption for specific tasks. In addition, often the experimental design allows for the monitoring of possible confounding factors. For this purpose, a basic analysis approach consists of fitting a linear model for the factors with the form:

$$Energy = c_{MF} \times MainFactor + c_{CF} \times CoFactor$$

The factor variables can be a basic indicator or continuous variables. The linear model will be subject to an ANalysis Of VAriance (ANOVA) to understand the statistical significance of the factor effects on the power. ANOVA is a statistical method to analyze the difference of means among different groups; ANOVA attributes the variance of means to different sources and evaluates the probability that an observed difference is due to an actual effect of factor versus random effects (e.g. measurement noise). Typically such

probability – called p-value – is compared against a predefined threshold (5% is a common choice) to decide whether it is possible to state that the treatment had a real effect.

The ANOVA is a parametric test, meaning that its results are reliable when a few conditions are met, the most important being the normality of the samples. The normality can be checked by means of the Shapiro-Wilk test; if the test return a a p-value smaller than a given α level it is possible to conclude that the data is not drawn from a normal distribution.

When the parametric assumption for ANOVA are not be met, a permutation test alternative to ANOVA can be used (e.g., using the lmPerm R package [13]). In addition to the statistical significance, it is important to evaluate the magnitude of the effect of the factors. A basic assessment can be performed by looking at relative values of the estimated regression coefficients or by means of standardised coefficients, such as $\eta 2$.

When a simple comparison of two samples is required, without any co-factor involved a t-test can be applied, being a simplified version of an ANOVA.

2.2.7. Threats to Validity Analysis

Regardless of the chosen approach, the How deliverable must contain an analysis of three different threats to validity.

Internal validity. It depends on whether the consumption data is related to the execution of the SWUT. Several possible cases include the following:

- The device has no operating system and executes only the SWUT. The consumption of the device can be attributed entirely to the SWUT.

- The device has a multitasking OS. The SWUT and other processes (at the application or OS level) run concurrently. The problem is how to attribute the consumption of the device to each process (and to the SWUT, in particular). An option is to stop all processes except the one that executes the SWUT. This is unfeasible in most OSs, so the remaining option is to minimise the set of running processes to those strictly required by the OS. Then, it is possible to measure the device consumption both when the device is idle, i.e., only OS-related processes are running, and when it is running the SWUT. The difference between the two consumption values represents a reasonable approximation of the effective consumption attributable to the SWUT.

- The device has a multicore processor. The SWUT can be executed on any core at a specific CPU frequency. For this reason, it is unlikely that two consumption measurements for the same SWUT performed on the same device provide the same value.

The execution of the SWUT not in isolation, might be less a threat when the goal of the process is to perform a comparison. In such a case, a comparison can be performed when assuming the noise produce by other programs is similar for all tested alternatives.

Construct validity. It depends on how consumption is measured as well as the precision of the measurement:

- Instant power consumption has precision impacted mostly by the precision of the current measurement, and by the noise produced by processes executed in parallel with the SWUT (see discussion above on internal validity).

- Time measurement has precision impacted by the measure of the energy contained by the battery, by the non-linear discharge pattern, by the reduction of battery capacity with the recharging cycles, and by the time required to identify that the energy contained by the battery falls below a defined threshold.

- Model estimation builds on the precision of the model as its key attribute. The model may or may not consider relevant factors (for instance, heating) and, therefore, produce poor estimates.

Conclusion validity. It is the final category of threats to analyse during this phase. For gaining statistical evidence, the researcher must plan a certain number of repetitions of the same consumption measurement. Sometimes this can be an issue, especially in time measurements where each run can last hours. Thus, when it is not feasible to plan many repetitions of the same run, the investigator should consider a tradeoff between the number of repetitions and the possible error in the conclusions. Appropriate statistical tests should be used to determine the likelihood of observed differences or the confidence intervals associated with measurements.

2.3. Phase III: Do

In this phase, the researcher implements the experiment designed in the previous phases. The crucial part is the procedure automation. Each execution of the procedure should be autonomous, and at the conclusion, the researcher should be able to collect the data without interventions. Human intervention will alter the procedure execution because it will not be repeatable with the same actions. Achieving this requires defining a script that performs the same procedure multiple times. So, the goal of this phase is to provide an automated procedure valid for the DUT(s) used in the experiment.

In the case of instant power consumption measurement, the scripts should automate both the data acquisition and the SWUT execution. When performing a time measurement, the script must store all the system times as well as manage the battery recharge to avoid human intervention. In [14] and [15], the authors explained a possible implementation of this kind of scenario automation for time measurements. For model measurements, the scripts run all the software measurements tools defined in the previous phase and collect resource usage logs for each scenario.

An incorrect setup of the experiment poses a threat to *Construct validity* of the results since it could lead to measuring the wrong construct.

The output of this phase will be the scripts, which automate the data collection procedure and a set of files, which contain the raw energy consumption data according to the data format provided in the previous phase. The Do Deliverable document, introduced in Table 1, will be a synthetic report that lists and explains the content of each script and raw data. The availability of scripts and data make the replication and verification of results – essential in any scientific approach – to be carried out by third party. A recommended practice, is to leverage open public repositories – e.g. figShare, Zenodo, and GitHub – to store scripts and data.

2.4. Phase IV: Analyse

In this phase, the consumption data collected in the previous phase is analysed. There are two approaches for identifying task-related data in power traces:

- Online with synchronisation between the recorder and under measurement systems, and
- Offline using added markups to the traces.

With the first approach, only the portion of the traces pertaining to the observed tasks is recorded and later processed. The approach requires accurate synchronisation that is based on the capability to timely communicate between the device and the measurement instrumentation.

The second approach requires all the traces for a series of experiments to be recorded, and then, during an analysis phase, the segments pertaining to the observed tasks are extracted and processed. It requires no synchronisation as it suffices for trivial instrumentation to add markups into the traces. This approach is supported by the R package Powtran[2]. The result of the power trace analysis is the total amount of energy consumed to perform a task.

The energy consumption obtained in either way can then be analyzed according to the method defined in the How phase.

The Analyse phase might pose a threat to the *Conclusion validity*. In particular the data must be checked for the presence of outliers, which must be assessed, then a decision must be taken concerning their possible removal. In addition the distribution of the energy data should be identified; this is important to allow the choice of the appropriate statistical tests.

The output of this phase is a deliverable, called the Analyse Deliverable as described in Table 1, which contains Data analysis scripts, the Data analysis results, and the conclusion threats to validity analysis.

3. Applying the Consumption Measurement Process

In this section, we show how the proposed process can be applied to an example in which a battery-powered Raspberry Pi is used to sort integer values gathered by a sensor. The experiment can be deemed as representative of a typical environment in which measuring the energy consumption of a software application is required, since that estimation is crucial for the development of embedded software [16]. In the example, it is required to choose the most efficient sorting algorithm to maximize battery time. Given that the issue is the battery time, all the consumption measurements will be energy measurements. The following subsection is a process deliverable according to our proposed framework.

[2] `https://github.com/SoftengPoliTo/powtran/` (Last Visited: 2019/09/22)

3.1. Goal Deliverable

As defined in Section 2.1 the Goal deliverable contains the research questions, the description of SwUT, device, context, and the external threats to validity analysis.

3.1.1. Research Questions

RQ1 Compare energy consumption of Counting Sort algorithm implemented in C language and Merge Sort algorithm implemented in C language run on Raspberry Pi version 2B in the context of Raspbian Linux OS:

- **RQ1a** Characterise the energy consumption of the Counting Sort algorithm implemented in C language run on Raspberry Pi version 2B in the context of Raspbian Linux OS;
- **RQ1b** Characterise the energy consumption of the Merge Sort algorithm implemented in C language run on Raspberry Pi version 2B in the context of Raspbian Linux OS.

3.1.2. SWUT Description

The following two SWUTs are considered in the experiment:

- Counting sort: 2-pass sorting algorithm, with *O(n)* time complexity;
- Merge sort: single-pass sorting algorithm, with *O(nlogn)]* time complexity.

A brief description of the considered algorithms and the code implementation are reported in appendix Appendix A.1.

We planned five distinct dataset to test the SwUT labeled with numbers from 1 to 5. The first dataset contains numbers from 0 to (DATASET_SIZE -1) in ascending order and the second dataset contains numbers in descending order from (DATASET_SIZE -1) to 0. The remaining three datasets contain pseudo-random numbers with values between 0 and (DATASET_SIZE-1). The seed is known, so the same pseudo-random numbers can be generated anytime. For these data, DATASET_SIZE represents 50000 elements.

3.1.3. Device Specifications and Context

The most relevant hardware specifications for the tested device, a Raspberry Pi 2B, are specified in table 4.

The context of the measurement is specified in table 5. The full set of device specifications and the complete list of processes running during the experiment is reported in appendix Appendix A.2.

Parameter	Value
CPU	900Mhz Quad-Core ARM Cortex-A7
RAM	1GB
Graphics Core	VideoCore IV

Table 4: Goal deliverable - Devices

Parameter	Value
OS	Raspbian Linux OS: Jessie Lite
Kernel Version	4.4
OS Config.	Default
No. running processes	22
Power information collection interface	ADC NI USB 6210
Power information processing	C software written with NI library[3]

Table 5: Goal deliverable - Context

3.1.4. Threats to External Validity Analysis

The results will be valid only for Raspberry Pi version 2B, and the experimenters accept this restriction.

3.2. How Deliverable

As defined in section 2.2, the how deliverable for an Instant Power measurement will contain the following sections: Hardware Instrumentation, Synchronization, Sampling Frequency, File Format, Threats to Validity Analysis.

3.2.1. Hardware Instrumentation

- Voltage generator: 5V (max 2A)
- Shunt Resistor: 0,05 Ω
- ADC: National Instrument NI-6210
- Supervising device: Desktop Computer

3.2.2. Sampling and Data Synchronization

There will be no clocks synchronisation or post-processing data analysis.

3.2.3. Sampling Frequency

The selected sampling frequency is 125 khz.

3.2.4. File Format

The file name includes the following details about the experiment: device maker, device model, algorithm name, programming language, dataset size, dataset label (e.g., progressive number). A sample file name can be *Raspberry_2b_counting_c_5000_1*. A file content sample can be the following:

```
1,149160E+0
1,142452E+0
1,152316E+0
```

3.2.5. Threats to Validity Analysis

To limit the Threats to Internal Validity related to the correct determination of the consumption value for a specific process, we plan to:

- Run the experiment on a new installation of a Raspbian Lite OS to minimise the number of concurrent processes,
- Measure the instant power consumption of the device in idle, and
- Subtract the idle value from the data obtained in each run (see section 3.2.7).

To limit the Threats to Construct Validity, we provide a voltage measurement of the shunt resistor. The value logged in the file is the voltage multiplied by the voltage divided by the shunt resistor value. This multiplication will provide the instant power consumption value according to Ohm's Law, $P = V\,I$. In this computation, we do not take into account the shunt resistor temperature, which could alter our measurement. We are willing to accept this error because it is not going to affect our results significantly.

To limit the Threats to Conclusion Validity, we will repeat each measurement 30 times.

3.2.6. Do Deliverable

We will collect instant power consumption during the execution of the algorithm. In the analysis phase, we will transform instant power consumption to an energy value by computing the integral of instant power consumption over the experiment time interval. For automating the experiment, we created a script in the Python language, to:

- Run the data collector on the supervising device;

- Run the SwUT on the Raspberry Pi;
- Store the instant power consumption on a text file;
- Commit and push the instant power consumption file to a local git repository.

The Raspberry Pi is connected to a router on a LAN. The supervising device is also connected to the same network, so it is possible to run the SwUT via SSH. The script reads the To-Do Listfrom an input file. The To-Do List includes a line representing each run of the SwUT specifying the following information:

- Device Model;
- Programming Language;
- SwUT;
- Dataset Size;
- Dataset Label.

To repeat these operations programmatically, we created a program, called executor, to run on the Raspberry Pi, which takes the following as input:

- The SwUT name;
- The Dataset size;
- The dataset Label.

When started, the executor will:

1. Create the dataset dynamically;
2. Run the marker;
3. Run the SwUT;
4. Repeat points 2 and 3 thirty times;
5. End.

The experiment automation script in python, the To-Do List file, and the raw data are available online on an open repository [17].

Each element of the To-Do List executes the same task thirty times, as described in Section 3.2.5. Each run is preceded by the implementation of a marker, which allows the identification of the SwUT in the instant power consumption data.

Table 6: Summary statistics of energy by algorithm

Algorithm	Mean	Median	SD	p.SW
counting sort	2.30	2.33	0.25	p <0.001
merge sort	8.69	8.55	1.22	p <0.001

3.2.7. Analyse Deliverable

Corresponding to the original RQ, we formulate the following null hypothesis: There is no significant difference in the central tendency of the energy consumed by the two algorithms in performing the sorting task. The significance level (α), corresponding to the risk of committing a type I error, i.e., rejecting the null hypothesis while it is true, may be assigned to the standard 5%.

Analysis Results. The distribution of the energy consumed per task can be represented graphically by means of a boxplot displayed in Figure 5

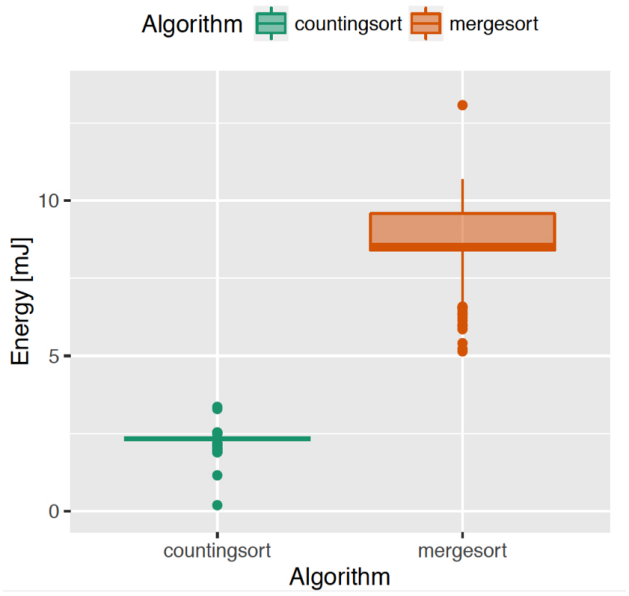

Figure 5: Energy consumed by the two algorithms for sorting an array of 50,000 elements.

A summary of the data, together with a central tendency, dispersion, and normality is reported in Table 6. The values are reported in millijoules.

We observe that the last column, reporting the p-value of the Shapiro-Wilk test, contains values that are smaller than 5% (our reference value). We can reject the null hypothesis for both algorithms that the values are sampled from a normal distribution. Therefore, we should apply non-parametric statistics in the following analysis. To check the null hypothesis, we can apply a Mann-Whiteny U test. The p-value returned by the test is smaller than the reference level, so we can reject the null hypothesis. We conclude that a significant difference in energy consumption exists between the two algorithms. To quantify the magnitude of the difference, we compute the standardized effect size. For this purpose, we adopt the Cliff's Delta statistic. We obtain an effect size of -1 meaning that the amount of energy consumed by the first algorithm (counting sort) is smaller than the second (merge sort) by a significant amount. So, we conclude counting sort is the algorithm to select for better energy efficiency.

4. Related Work

During recent years, the interest in how software influences the power consumption of a device has increased sharply. It is possible to divide the related work on the topic into two categories:

1. Energy consumption measurement/estimation.
2. Energy consumption reduction/optimization.

The first category focuses on the way in which energy is measured or estimated. A recent work by Harman et al. [7] categorizes Energy Testing as one of the most important fields for Search-Based Software Engineering, and highlights the need for trustable metrics and for quick and well-defined energy-measuring procedures. The paper also highlights several novel hardware-based approaches, e.g., the SEEP [18] approach using symbolic execution to capture and re-execute paths. The approach we propose is adaptable to any alternative method for measuring energy or power, since using a different procedure would only have impact on the hardware section of the How deliverable and on the Do deliverable where the steps of the experiments are formalized.

Noureddine et al. [19] review different energy measurement approaches that can be classified as measurement/estimation and modelling. In this first sub-category, the goal is to determine the energy consumption through the hardware equipment, while the latter creates a mathematical model of the

device energy consumption to provide energy data without external equipment. By analysing the literature, we see that Hindle et al. [20] proposed an approach to measure how the energy consumption of software applications varies through the different versions. There exist working prototypes, which allow estimating the energy consumption of mobile devices, the most popular being DevScope [21], AppScope [22], and En-Track [23]. This work proposes complete and working prototypes for measuring the power consumption of Android applications. The main problem of these approaches is the limited number of supported devices. For this reason, it is difficult to replicate the studies to validate the measurements or to apply the same measurement on a slightly different device or software.

The second category focuses on the changes to be made to the architecture or the source code to achieve the energy consumption reduction or optimisation. The literature review by Aleti et al. [24] describes some approaches to reduce the energy consumption by improving the software architecture.

All these efforts are typically individual optimisation and are difficult to apply to general cases. Furthermore, both categories share some common steps to be performed, such as data collection, code instrumentation, and data analysis, but often it is not easy to compare the procedures in different experiments since a uniform notation for the documentation of similar tasks is missing. So, it is useful to think about a general process to measure the power consumption of a software application, and to provide the tools needed to document and analyse the data obtained in the measurement. To our knowledge, such a general and repeatable approach is still missing in the literature.

It is possible to identify many references that measure the energy consumption of software applications and propose ways to reduce it, such as [9] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40], [41]. In Table 7 we compare the information (listed as table columns) provided by our process along with the information provided by each of these papers (listed as table rows). A check mark (✓) indicates that the current information carried out by our process is also included in the related paper. In Section 2 we explain in detail all information produced as the output of our process. Table 7 shows that in the literature there are methods for measuring the energy consumption of software applications. However, there is no common procedure to extract the energy data from software applications. In detail, all the analysed works lack the following features:

Table 7: Comparison between related work and our process

Related paper	[9]	[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]	[38]	[39]	[40]	[41]
Goal Deliverable																		
RQ Definition	✓		✓		✓	✓		✓	✓	✓		✓	✓		✓	✓	✓	✓
Software Under Test Description	✓			✓				✓		✓	✓		✓	✓	✓		✓	
Device Context Info	✓		✓					✓			✓		✓			✓		✓
How Deliverable																		
Measurement or Estimation Technique Description	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Hardware and Software Instrumentation			✓		✓								✓			✓		✓
Sampling Frequency Used						✓	✓		✓	✓		✓		✓	✓		✓	
Data Format Description																		
Do Deliverable																		
Implementation Scripts Description and Publication																		
Raw Data Publication																		
Analyse Deliverable																		
Statistical Data Analysis	✓				✓	✓				✓		✓	✓		✓		✓	
Threats to Validity Analysis	✓				✓	✓			✓	✓		✓	✓		✓	✓	✓	✓

- Provide all the information that are part of our process;
- Explain step-by-step how to replicate the experiment, and
- Provide a defined format to publish the raw data obtained.

Following a defined procedure will enable a comparison between data of different experiments. This will guide developers toward countermeasures to handle cases of high-energy consumption. We previously identified a high-level framework [42], which describes the motivations that lead to measuring the energy consumption of software applications. So, the main contribution of this work proposes a common process to be used by anyone to extract energy data of software applications in such a way as to have comparable data that is extracted and analyzed in a standard way.

5. Conclusion

The awareness of energy consumption is an emerging quality for software and hardware. The expansion of mobile device usage as well as the diffusion of IoT devices made energy consumption a critical issue due to the limited amount of energy batteries can store. In this paper, we presented a well-defined and rigorous approach to plan and conduct software energy consumption measurements that, to the best of our knowledge, was not previously available in the literature. The proposed procedure incorporates features enabling the adoption of evidence-based software engineering as it produces results that are:

- Trustable: detailed documentation of goals, planning, and execution allows quality assessment.
- Comparable: the contextual details and the uniformity of the process ease comparison.
- Actionable: the factors are defined and, thus, any energy improvement actions can be properly targeted.

The approach is applicable in a real-world context and has been applied by the authors in previous research. In addition, a sample application is reported to serve as a template guide for third-party applications.

Furthermore, the approach also serves as a checklist for assessing existing studies. We used it in this sense to evaluate the related work, as summarized in Table 7.

For future work, we plan to create a repository where it will be possible to upload the deliverables produced according to the process we describe. Such repository would allow comparing different studies and building an empirically backed body of knowledge.

6. Statements

6.1. Conflict of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

6.2. Data Availability

The data used to support the findings of this study are included within the article in the form of references linking to resources available on the figShare public open repository.

References

[1] J. Bornholt, T. Mytkowicz, K. S. McKinley, The model is not enough: Understanding energy consumption in mobile devices, in: Proceedings of 2012 IEEE Hot Chips 24 Symposium (HCS), pp. 1–3.

[2] P. Fairley, Blockchain world - feeding the blockchain beast if bitcoin ever does go mainstream, the electricity needed to sustain it will be enormous, IEEE Spectrum 54 (2017) 36–59.

[3] B. Mochocki, K. Lahiri, S. Cadambi, Power analysis of mobile 3d graphics, in: Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings, DATE '06, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006, pp. 502–507.

[4] T. Dyba, B. A. Kitchenham, M. Jorgensen, Evidence-based software engineering for practitioners, IEEE Software 22 (2005) 58–65.

[5] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[6] R. Van Solingen, V. Basili, G. Caldiera, H. D. Rombach, Goal question metric (GQM) approach, Encyclopedia of software engineering (2002).

[7] M. Harman, Y. Jia, Y. Zhang, Achievements, open problems and challenges for search based software testing, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, pp. 1–12.

[8] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, Y.-M. Wang, Fine-grained power modeling for smartphones using system call tracing, in: Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, ACM, New York, NY, USA, 2011, pp. 153–168.

[9] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, A. De Lucia, Software-based energy profiling of android apps: Simple, efficient and reliable?, in: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), IEEE, pp. 103–114.

[10] A. A. Nacci, F. Trovò, F. Maggi, M. Ferroni, A. Cazzola, D. Sciuto, M. D. Santambrogio, Adaptive and flexible smartphone power modeling, Mobile Networks and Applications 18 (2013) 600–609.

[11] R. Hogg, E. Tanis, Probability and Statistical Inference, Prentice Hall, 2006.

[12] R. Saborido, V. Arnaoudova, G. Beltrame, F. Khomh, G. Antoniol, On the impact of sampling frequency on software energy measurements, PeerJ PrePrints 3 (2015) e1219.

[13] B. Wheeler, M. Torchiano, lmPerm: Permutation Tests for Linear Models, 2016. R package version 2.1.0.

[14] L. Ardito, M. Torchiano, M. Marengo, P. Falcarin, glcb: an energy aware context broker, Sustainable Computing: Informatics and Systems 3 (2013) 18 – 26.

[15] L. Ardito, Energy aware self-adaptation in mobile systems, in: Proceedings - International Conference on Software Engineering, pp. 1435–1437.

[16] M. Ibrahim, M. Rupp, H. Fahmy, A precise high-level power consumption model for embedded systems software, EURASIP Journal on Embedded Systems 2011 (2011) 480805.

[17] R. Coppola, M. Torchiano, L. Ardito, Methodological Guidelines for Measuring Energy Consumption of Software Applications - Replication Package for Raspberry PI case study, `https://doi.org/10.6084/m9.figshare.9879503.v1`, 2019.

[18] T. Hönig, C. Eibel, R. Kapitza, W. Schröder-Preikschat, Seep: exploiting symbolic execution for energy-aware programming, in: Proceedings of the 4th Workshop on Power-Aware Computing and Systems, ACM, p. 4.

[19] A. Noureddine, R. Rouvoy, L. Seinturier, A review of energy measurement approaches, SIGOPS Oper. Syst. Rev. 47 (2013) 42–49.

[20] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, S. Romansky, Greenminer: A hardware based mining software repositories software energy consumption framework, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 12–21.

[21] W. Jung, C. Kang, C. Yoon, D. Kim, H. Cha, Devscope: A nonintrusive and online power analysis tool for smartphone hardware components, in: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '12, ACM, New York, NY, USA, 2012, pp. 353–362.

[22] C. Yoon, D. Kim, W. Jung, C. Kang, H. Cha, Appscope: Application energy metering framework for android smartphones using kernel activity monitoring, in: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 36–36.

[23] S. Lee, W. Jung, Y. Chon, H. Cha, Entrack: A system facility for analyzing energy consumption of android system services, in: Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '15, ACM, New York, NY, USA, 2015, pp. 191–202.

[24] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, Software architecture optimization methods: A systematic literature review, IEEE Transactions on Software Engineering 39 (2013) 658–683.

[25] C. Seo, S. Malek, N. Medvidovic, An energy consumption framework for distributed java-based systems, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 421–424.

[26] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: 2012 First International Workshop on Green and Sustainable Software (GREENS), pp. 55–61.

[27] S. Islam, A. Noureddine, R. Bashroush, Measuring energy footprint of software features, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp. 1–4.

[28] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, ACM, New York, NY, USA, 2014, pp. 36:1–36:10.

[29] D. Li, S. Hao, W. G. J. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, ACM, New York, NY, USA, 2013, pp. 78–89.

[30] S. Hao, D. Li, W. G. J. Halfond, R. Govindan, Estimating mobile application energy consumption using program analysis, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 92–101.

[31] D. Li, W. G. J. Halfond, An investigation into energy-saving programming practices for android smartphone app development, in: Proceedings of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, ACM, New York, NY, USA, 2014, pp. 46–53.

[32] D. Li, S. Hao, J. Gui, W. G. J. Halfond, An empirical study of the energy consumption of android applications, in: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 121–130.

[33] D. Li, Y. Jin, C. Sahin, J. Clause, W. G. J. Halfond, Integrated energy-directed test suite optimization, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 339–350.

[34] C. Sahin, F. Cayci, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Towards power reduction through improved software design, in: 2012 IEEE Energytech, pp. 1–6.

[35] D. Li, Y. Jin, C. Sahin, J. Clause, W. G. J. Halfond, Integrated energy-directed test suite optimization, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 339–350.

[36] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, J. Clause, How does code obfuscation impact energy usage?, in: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ICSME '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 131–140.

[37] S. Hao, D. Li, W. G. J. Halfond, R. Govindan, Estimating android applications' cpu energy usage via bytecode profiling, in: Proceedings of the First International Workshop on Green and Sustainable Software, GREENS '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 1–7.

[38] I. Manotas, L. Pollock, J. Clause, Seeds: A software engineer's energy-optimization decision support framework, in: Proceedings of the 36th

International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 503–514.

[39] K. Liu, G. Pinto, Y. D. Liu, Data-oriented characterization of application-level energy optimization, in: A. Egyed, I. Schaefer (Eds.), Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 316–331.

[40] K. Liu, G. Pinto, Y. D. Liu, Data-oriented characterization of application-level energy optimization, in: A. Egyed, I. Schaefer (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 316–331.

[41] D. Li, S. Hao, W. G. J. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: In ISSTA, pp. 78–89.

[42] L. Ardito, G. Procaccianti, M. Torchiano, A. Vetrò, Understanding green software development: A conceptual framework, IT Professional 17 (2015) 44–50.

[43] R. Coppola, L. Ardito, M. Torchiano, Methodological Guidelines for Measuring Energy Consumption of Software Applications - Acqusition Software, `https://doi.org/10.6084/m9.figshare.9879569.v1`, 2019.

Appendix A. Experiment Details

Appendix A.1. SWUT code

Counting sort is a 2-pass sort algorithm that is efficient when the number of distinct keys is small compared to the number of items. The first pass counts the occurrences of each key in an auxiliary array, and then makes a running total so each auxiliary entry is the number of preceding keys. The second pass puts each item into its final place according to the auxiliary entry for that key. Time complexity is *O(n)*. The implementation has been tested and follows the state of the art:

```
void counting_sort(int A[], int n) {
    int i, *B,*C;
    B = malloc(n * sizeof(int));
    C = malloc(M * sizeof(int));
    for (i=0; i<M; i++)
        C[i] = 0;
    for (i=0; i<n; i++)
        C[A[i]]++;
    for (i=1; i<M; i++)
        C[i] += C[i-1];
    for (i=n-1; i>=0; i--) {
        B[C[A[i]]-1] = A[i];
        C[A[i]]--;
    }
    for (i=0; i<n; i++)
        A[i] = B[i];
}
```

The Merge sort algorithm divides the items to be sorted into two groups, recursively sorts each group, and merges them into a final, sorted sequence. Time complexity is *O(nlogn)*. The implementation has been tested and follows the state of the art:

```
void my_merge_c(int *v, int dim) {
    int *aux;
    aux = (int *) malloc (dim * sizeof(int));
    merge_sort_recur(v,0, dim-1, aux);
}
```

```
981 void merge_sort_recur(int *v, int p, int r,
982                        int *aux) {
983     int q;
984     if (p < r) {
985         q = (p+r)/2;
986         merge_sort_recur(v,p,q,aux);
987         merge_sort_recur(v,q+1,r,aux);
988         my_merge(v,p,q,r,aux);
989     }
990 }
991
992 void my_merge(int *v, int p, int q, int r,
993               int *aux) {
994     int i, j, k;
995     for ( i=p, j = q+1, k = p;
996                         i<=q && j<=r; ) {
997         if (v[i] < v[j] )
998             aux[k++] = v[i++];
999         else
1000            aux[k++] = v[j++];
1001    }
1002    while (i <= q)
1003        aux[k++] = v[i++];
1004    while (j<=r)
1005        aux[k++] = v[j++];
1006    for(k=p; k<=r; k++)
1007        v[k] = aux[k];
1008 }
```

1009 *Appendix A.2. Devices and Context*

1010 The hardware specifications for the tested device, Raspberry Pi 2B, in-
1011 clude:

1012 - A 900MHz quad-core ARM Cortex-A7 CPU
1013 - 1GB RAM
1014 - USB ports: no devices connected

- 40 GPIO pins: not used for our experiment
- Full HDMI port: no display connected
- Ethernet port: connected to a local router without Internet connection
- Combined 3.5mm audio jack and composite video: not used
- Camera interface: not used
- Display interface: not used
- Micro SD: Kingston 16GB Class 10
- VideoCore IV 3D graphics core

For this experiment, the context may be summarised as follows:

- Raspbian Linux OS: Jessie Lite, Kernel version 4.4
- Default OS configuration
- Processes running during the experiment:
  - kworker
  - systemd
  - kthreadd
  - ksoftirqd
  - rcu_sched
  - rcu_bh
  - migration
  - kdevtmpfs
  - netns
  - perf
  - khungtaskd
  - writeback
  - crypto

  - bioset
  - kblockd
  - rpciod
  - kswapd0
  - vmstat
  - fsnotify_mark
  - nfsiod
  - kthrotld
  - bioset

- Power information collected through ADC NI USB 6210
- Power information processed through custom software written in the C language using the default NI library. The software has been made available online through an open repository [43].