Master Thesis
at the Department of Information Technology
and Electrical Engineering

Powerful Software

AS 2013

Etienne Geiser

Advisors: Pratyush Kumar
Lars Schor
Professor: Prof. Dr. Lothar Thiele

Zurich
21st February 2014

Abstract

Attempting to contribute to the ever increasing energy efficiency discussion, this master thesis proposes a software-based method to approximate energy consumption on battery-powered devices. The only inputs it depends on are the battery voltage and the battery capacity. For power discussions, time measurements are additionally needed. In a calibration step, we create a battery voltage to charge mapping, the so-called discharge curve. The discharge curve and the voltage at the beginning and at the end of the interval of interest is then used to calculate the energy consumption during that interval.

Further, we derive error boundaries for the different stages of our energy approximation. In the error analysis we focus on errors originating from the inaccuracy of the voltage measurement and the usage of a discrete discharge curve.

After implementing our method on an Android device, we evaluate its effectiveness in various case studies. For the evaluation we wrote an application, which periodically measures the energy consumption. We then use our application to evaluate general-purpose and specific applications for their energy consumption. In particular, we compare various sorting algorithms against each other and concluded which are the more energy efficient ones.

Acknowledgements

I would like to express my gratitude to my supervisors, Lars Schor and Pratyush Kumar for their time, ideas, constructive remarks and engagement throughout the learning process of this master thesis. Furthermore, I thank Prof. Dr. Lothar Thiele for giving master students the opportunity to conduct research within the realms of the research group. Last, but not least, a big thank you goes to my family, for their unfaltering support.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Background 1
1.3 Contribution 2
1.4 Overview 2

2 Approach **3**
2.1 Flow of Approach 3
2.1.1 Reading the Voltage File 4
2.1.2 Discharge Curve 4
2.1.3 Energy Calculation 5
2.1.4 Linear Interpolation 6
2.2 Accuracy 8
2.2.1 Measurements 8
2.2.2 Interpolation 8
2.2.3 Discharge Curve 9
2.2.4 Charge Approximation 10
2.2.5 Energy Estimation 11

3 Evaluation **13**
3.1 Measurement Setup 13
3.2 Evaluation 15
3.2.1 Power Evaluation 15
3.2.2 Discharge Curve 17
3.2.3 Charge Estimation 18
3.2.4 Differentiating Components 20
3.2.5 Comparison of Different Sorting Algorithms 21
3.2.6 Evaluation of Common Applications 23

4 Conclusion and Outlook **26**
4.1 Conclusion 26

4.2 Outlook 27

A Examples of Sorting Algorithms **30**

B Algorithm Code **33**

B.0.1 Support Functions 33

B.0.2 Bubble Sort 33

B.0.3 Insertion Sort 34

B.0.4 Quick Sort 35

C Data Tables for Sorting Algorithms **39**

D Presentation Slides **41**

List of Figures

2.1 Flow Discharge Curve 4
2.2 Charge Difference Approximation 6
2.3 Interpolation Overview 7
2.4 Interpolation Worst Cases 9
2.5 Accuracy of Charge Approximation 11
2.6 Minimal and Maximal Charge Difference 12

3.1 Wire-Bridge 15
3.2 Voltage over Multimeter 16
3.3 Voltage over Multimeter Averaged 17
3.4 Discharge Curve 18
3.5 Merged Discharge Curve 19
3.6 Effect of Discharge Curve Slope 19
3.7 Charge Estimation 20
3.8 Insertion versus Screen 21
3.9 Bubble Sort and Insertion Sort Evaluation 23
3.10 Merge Sort and Quick Sort Evaluation 23
3.11 Comparison of Common Applications 24
3.12 Youtube Movie Voltage 25

1 Introduction

1.1 Motivation

Energy efficiency is given more attention than ever before. Not only in terms of green thinking for saving our environment, or in terms of profit as less energy used means more money earned, but also simply in terms of user comfort. Who does not own a mobile electrical device today? For a mobile device less energy consumption means longer independence of the need to recharge the battery. With higher energy efficiency new features can be added to our device, features which we could not add previously because their power consumption was too high, draining the battery too fast. How can the energy consumption be decreased? Both the hardware and the software can be improved. While a programmer generally cannot take much influence on the hardware, he can aim to write his code in such a fashion that it consumes as little power as possible.

This thesis presents a tool for programmers to analyse the energy consumption of their programs, and subsequently to optimize them.

1.2 Background

As a target platform for this thesis, we choose Android smart phones due to being a common mobile device with an open source operating system. There are already applications, which give an estimate of the power consumption

of applications running on a device. We would particularly like to mention *PowerTutor [1]*. At the outset of the thesis we initially attempted to use it to measure the power consumption. However, realized that *PowerTutor* estimates the consumption by dividing the phone into hardware components. It then measures the time that an application uses these components and multiplies each time with a component specific coefficient. The drawback of this approach is that it is necessary to know the components and their specific coefficients, making it hardware dependent, which greatly reduces the flexibility. For calculating these component coefficients one would need to be able to measure the power the components consume. Therefore, we decided to develop our own software-based method.

1.3 Contribution

In this thesis we prove that it is possible to create a tool capable of giving an estimation of the power consumed without the need of knowing hardware specific consumption coefficients. This is done by proposing a software-based method to measure the energy consumption for battery-powered devices. Further we derive error boundaries for our energy approximations, addressing inaccuracies. After implementing the method on an Android device, we evaluate the effectiveness of the method in various case studies.

1.4 Overview

Following on the introduction, Chapter 2 explains in the first part our approach and presents thereafter in the second part an analysis of the error estimation. The third chapter focuses on putting our approach to test in case studies.

2
Approach

This chapter discusses our approach of obtaining the amount of energy consumed in an Android device. In the first part, we present the approach, in the second part, we discuss the accuracy of our method.

2.1 Flow of Approach

On Android devices a common approach to estimate the power consumption is to measure the time during which a component (e.g. CPU, GPU, ...) is used. An example of a tool based on this approach would be *PowerTutor*. This approach has the drawback, that it requires knowledge about all the components and their power consumption. As the components and their consumptions often vary, a list of all components and their specific consumption for each phone type and version is required. Our aim though was to have a tool, which can be applied independent of the device. Thus we looked for a more general, software-based approach and settled for the following: At the centre of our testing stage lies the voltage measured across the battery. In the batteries used today, the battery voltage decreases with decreasing charge. This voltage to charge relation over the whole charge range is called the discharge curve. The discharge curve allows us to calculate the charge remaining in the battery, using the voltage measured over the battery. For our purpose we assume that the discharge curve changes only little over time, which means we can neglect the so-called ageing process, alleviating the need to regularly update the discharge curve used for converting

voltage/charge values. With the ability to estimate the charge from voltage and therefore knowing how much energy is stored in the battery, we are then able to estimate the energy used between two voltage measurements. The energy consumed is the difference between the charges we calculated from the voltages, multiplied by the total battery capacity. In the following subsections we introduce the different stages of our method.

2.1.1 Reading the Voltage File

The voltage is acquired by reading the system-provided file found at the path */sys/class/power_supply/battery/voltage_now*. It contains the battery voltage measured in µV. The smallest voltage difference between two measurements we observed is 1250µV, indicating that this is the granularity of the voltage meter. Our measurements also indicate, that this file is updated about every 50 seconds. This has the drawback, that measuring the voltage on a shorter time interval often returns the same voltage, and with no voltage difference we can not calculate the energy consumption.

2.1.2 Discharge Curve

Once the process of obtaining the current voltage is established, our next step is to create a discharge curve. A discharge curve can be generated by draining the battery at a constant rate from 100% charge to 0% charge. Due to the constant power consumption, we know that after 60% of the total time it took for the complete discharge, we have consumed 60% of the total battery capacity. This behaviour allows us to label the *time* axis as a *relative charge used* axis.

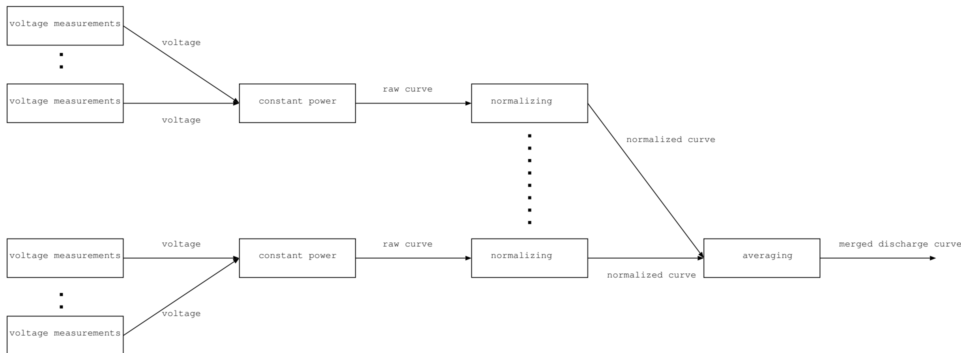


Figure 2.1: Flow Discharge Curve
Above we see the flow of creating a merged discharge curve

In order to reduce noise and improve accuracy, we use multiple discharge curves, normalize them and then merge them into a single discharge curve.

Figure 2.1 outlines the flow of the creation of such a merged discharge curve. Looking at the measurement files we observe that the first and last voltages are not the same for different measurements taken. This is mainly due to the fact that the battery is not charged to a constant value, but rather charged to a specific value and then the charging is suspended until the voltage drops below a certain level. Therefore it is possible to start measuring at different voltage levels, even after "fully" charging the battery. Also, the time between unplugging the charger and starting the measurement was not strictly controlled. Due to varying starting voltages and the fact that we used various measurement periods the last voltage read may differ.

To avoid those issues, we normalize the individual curves relative to a specific voltage interval (3.6-4.05 V). In the interval we sample the voltage at every $\frac{1}{500}\%$ charge step.

A point or sample (C,V) in the discharge curve is given by a charge used (C) and the voltage (V). Such a point describes the voltage after a certain relative amount of battery charge has been consumed. Now, we average the voltages at these points over all counterparts of the different curves, resulting in the points of our averaged (or merged) discharge curve. Points outside the voltage interval are processed similarly, adding them up and then dividing them by the number of curves contributing to that point. Due to being discrete, the individual curves often do not have the exact values for the desired relative charge, thus the voltages for the desired charge are linearly interpolated values.

2.1.3 Energy Calculation

For an estimation of the energy consumed over a certain time interval, we log the voltage at the beginning (v_s) and at the end(v_e). Then we use our discharge curve to obtain the relative charges (c_s,c_e) for the two voltages. Refer to Figure 2.2. Because our discharge curve is a discrete curve, we usually have to use linear interpolation for voltage values between two discharge curve values. The difference between the two relative charges (Δc) represents the percentage which was used in comparison to the total capacity of the battery. Figure 2.2 shows an example of how the discharge curve is applied for calculating the charge difference. To get the amount of energy consumed (e), we multiply the charge difference with the total amount of energy (E) the battery provides during a single full discharge. Written as a formula:

$$e = (c_e - c_s) * E = \Delta c * E \tag{2.1}$$

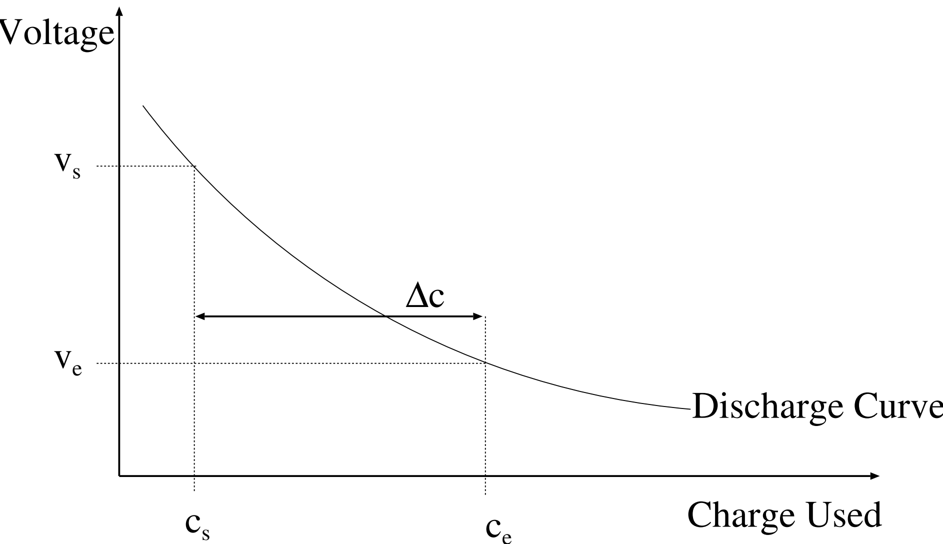


Figure 2.2: Charge Difference Approximation
The input voltages are converted to charges using the discharge curve.
Afterwards the difference of the charges are calculated

2.1.4 Linear Interpolation

This subsection elucidates the linear interpolation we use for estimating values, which our discrete discharge curve does not provide. Interpolation is required because our discharge curve consists only of an array of voltage-charge pair samples. Therefore we need an approximation for values in between two such samples. We assume that the discharge curve is monotonically falling, since a voltage increase would mean an increase in battery charge. This would contradict our assumption of a constant energy decrease during the creation of our discharge curve. The interpolation is used for approximating both, voltage and charge.

If we have the voltage v and want to approximate c from it, we choose the two samples (C_1,V_1),(C_2,V_2) from the discharge curve. The samples are chosen such that their voltages V_1 and V_2 encase our voltage v. See Figure 2.3 for a visualisation. V_1 is the next higher and V_2 the next smaller voltage found in the discharge curve.

At the base of the interpolation lies the Intercept Theorem [2, p. 16], which is in our case formulated as:

Intercept Theorem:

$$\frac{v_x}{c_x} = \frac{\Delta V}{\Delta C} \tag{2.2}$$

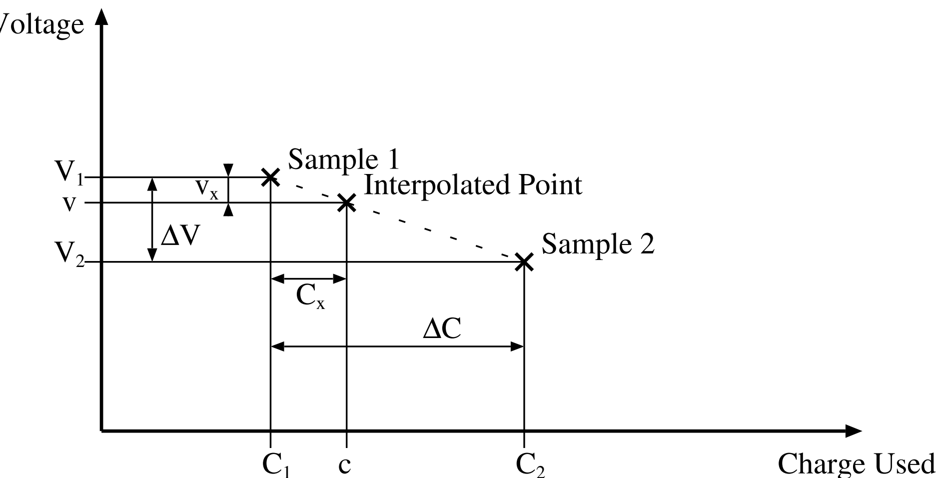


Figure 2.3: Interpolation Overview
Visualisation of an interpolation

Where we use the following notation:

$$v_x = V_1 - v \tag{2.3}$$
$$c_x = C_1 - c \tag{2.4}$$
$$\Delta V = V_1 - V_2 \tag{2.5}$$
$$\Delta C = C_1 - C_2 \tag{2.6}$$

Figure 2.3 puts them graphically into relation.

Used Charge Interpolation

For obtaining the formula for a charge approximation we first solve (2.4) to c.

$$c = C_1 + c_x$$

Then we apply (2.2) to c_x, which results in:

$$c = C_1 + \Delta C \frac{v_x}{\Delta V}$$

Finally we substitute v_x, ΔV, ΔC with (2.3),(2.5),(2.6), giving us the interpolation formula for c:

$$c = C_1 + (C_2 - C_1) \frac{V_1 - v}{V_1 - V_2} \tag{2.7}$$

Voltage Interpolation

To obtain the formula for (v) we follow similar steps as we did for c:

$$v = V_1 - v_x = V_1 - \Delta V \frac{c_x}{\Delta C}$$
$$v = V_1 - (V_1 - V_2)\frac{c - C_1}{C_2 - C_1} \tag{2.8}$$

2.2 Accuracy

In the following section we discuss the accuracy of our approach. Beginning with the measurements, we derive lower and upper error bounds for the different stages of our method.

2.2.1 Measurements

The reading of the voltage files returns values which have the smallest step size of 1250 µV, meaning that the exact value of v would be between $v - \frac{1250\mu V}{2}$ and $v + \frac{1250\mu V}{2}$. Thus we define our lower and upper bound for our voltage measurements as followed:

$$v_{min} = v - 625\mu V \tag{2.9}$$
$$v_{max} = v + 625\mu V \tag{2.10}$$

2.2.2 Interpolation

To reach a voltage higher than V_1, voltage v would need to increase after the first sample. For a voltage lower than V_2 v would need to increase from that lower value to V_2. Both these cases contradict our assumption of a monotonic falling discharge curve, so we can assume the voltage to be bounded by V_1 and V_2.

Figure 2.4 illustrates the two cases where the linear approximation returns the worst results.

These are when the voltage of the discharge curve between sample 1 and sample 2 immediately drops from V_1 to V_2 or stays at V_1 almost until C_2 is reached. Linearly approximating the voltage in Worst Case 1 at $c = C_1$ would return V_1, however the true voltage would be V_2. Or approximating the charge used at $v = V_1$ in Worst Case 2, would return C_1 instead of C_2.

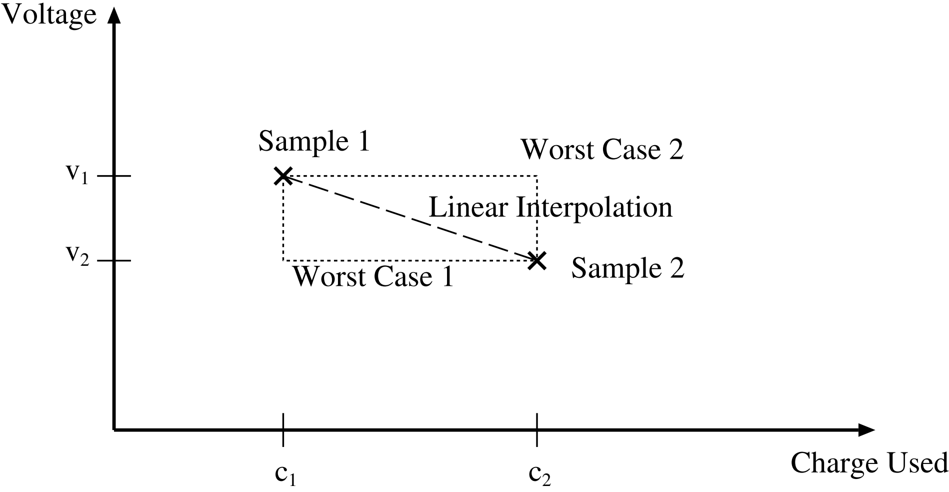


Figure 2.4: Interpolation Worst Cases
The dotted lines represent the two cases where the interpolation returns the worst results

We can conclude that:

$$V_2 < v < V_1$$
$$C_1 < c < C_2$$

Therefore, we can define the bounds for the linear interpolation as:

$$v_{min} = V_2 \tag{2.11}$$
$$v_{max} = V_1 \tag{2.12}$$
$$c_{min} = C_1 \tag{2.13}$$
$$c_{max} = C_2 \tag{2.14}$$

2.2.3 Discharge Curve

The formula for a voltage point (v_d) on the normalized and merged discharge curve is the following:

$$v_d = \frac{\sum_{n=1}^{M} v_n}{M}$$

Where v_n is the normalized point of the n-th curve used for creating our discharge curve. M represents the amount of different discharge curves over which v_d is averaged. Because the normalized point is an interpolation of

a voltage measurement, we use the previously derived bounds for measurements and linear interpolation for obtaining the bounding values of a discharge curve point.

This way we can define the lower bound of a discharge curve point as:

$$\begin{aligned} v_{d_{min}} &= \frac{\sum_{n=1}^{M} v_{n_{min}}}{M} = \frac{\sum_{n=1}^{M} V_{n_2}}{M} = \frac{\sum_{n=1}^{M} (V_{n_2} - 625\mu V)}{M} \\ &= \frac{\sum_{n=1}^{M} V_{n_2}}{M} - 625\mu V \end{aligned} \tag{2.15}$$

We first substitute $v_{n_{min}}$ with the lower bound of its interpolation using (2.11). Then we apply the lower bound of measurements (2.9) to V_{n_2}.

The upper bound can be derived similarly to the lower bound:

$$v_{d_{max}} = \frac{\sum_{n=1}^{M} V_{n_1}}{M} + 625\mu V \tag{2.16}$$

From these bounds it can be observed, that each point of the discharge curve has a different accuracy interval, making it impossible to give an accuracy formula for the entire discharge curve. Thus we save an additional curve for each the upper and the lower bounding points. This allows us to estimate an error bound during the usage of the discharge curve.

2.2.4 Charge Approximation

Approximating the used charge depends on two inputs with an inaccuracy. The first one is the voltage (v), which we want to convert into a charge (c). The second is the discharge curve we use to estimate the charge.

Figure 2.5 shows a graph with the upper ($Maximal\ Discharge\ Curve$) and lower ($Minimal\ Discharge\ Curve$) bound of the discharge curve, intersected by the upper (v_{max}) and lower (v_{min}) bound of the voltage.

We derive from Figure 2.5 the combinations of inputs which give a maximal (c_{max}) or minimal (c_{min}) charge used:

- c_{max} : Approximation with v_{min} using the $Maximal\ Discharge\ Curve$
- c_{min} : Approximation with v_{max} using the $Minimal\ Discharge\ Curve$

Due to the discrete discharge curve, the charge value is approximated by an interpolation, adding an additional inaccuracy. Using the bound for charge interpolation which we derived earlier, we can define upper and lower bounds for charge approximations:

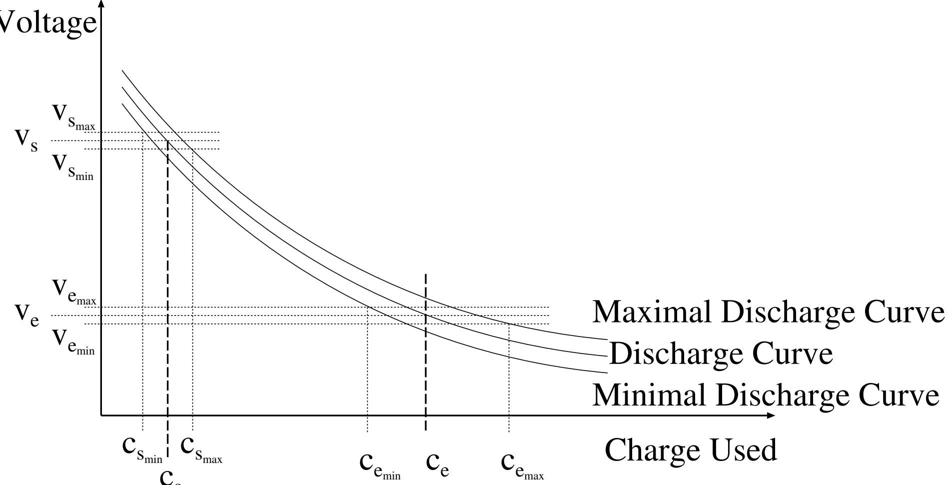


Figure 2.5: Accuracy of Charge Approximation
We can see which combinations of voltages and discharge curves result in the biggest ($c_{x_{max}}$) or smallest ($c_{x_{min}}$) respective charges

- $c_{max} = C$ of the (C, V) pair of the *Maximal Discharge Curve*, which has the biggest voltage smaller than v_{min}

- $c_{min} = C$ of the (C, V) pair of the *Minimal Discharge Curve*, which has the smallest voltage bigger than v_{max}

Figure 2.6 shows the additional inaccuracies which come from the interpolation.

2.2.5 Energy Estimation

In the formula for energy estimation (2.1) we see, that the energy depends on two charges.

Applying the bounds for charge approximation derived in Section 2.2.4 to those two relative charges, we can specify:

- $c_{e_{max}} = C$ of the (C, V) pair of the *Maximal Discharge Curve*, which has the biggest voltage smaller than $v_{e_{min}}$
- $c_{s_{min}} = C$ of the (C, V) pair of the *Minimal Discharge Curve*, which has the smallest voltage bigger than $v_{s_{max}}$
- $c_{e_{min}} = C$ of the (C, V) pair of the *Minimal Discharge Curve*, which has the smallest voltage bigger than $v_{e_{max}}$

- $c_{s_{max}} = C$ of the (C, V) pair of the $Maximal\ Discharge\ Curve$, which has the biggest voltage smaller than $v_{s_{min}}$

The maximal energy is reached when for the starting the smallest charge used value is subtracted from the biggest charge used at the end.
$e_{max} = (c_{e_{max}} - c_{s_{min}}) * E = \Delta c_{max} * E$

Where it is the opposite for the minimal energy:
$e_{min} = (c_{e_{min}} - c_{s_{max}}) * E = \Delta c_{min} * E$

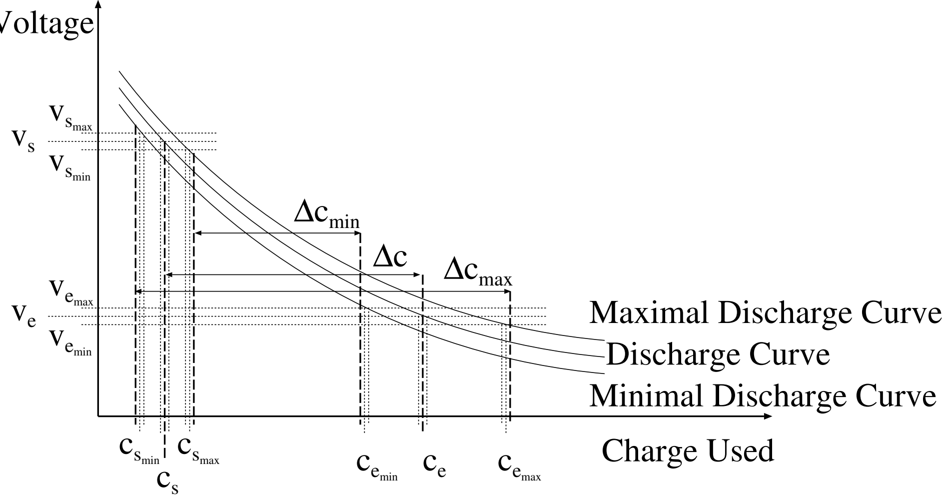


Figure 2.6: Minimal and Maximal Charge Difference
We can see the case when the charge difference reaches the maximal and when it reaches the minimal values

3
Evaluation

In the preceding chapter we laid down the theoretical foundation and developed our method. This chapter focuses on testing the practical usage of the method. The first section elaborates on the setup for our measurements, including the devices and tools used. The second section discusses different case studies we have conducted.

3.1 Measurement Setup

We wanted the device on which we conduct our experiments to be a standard, commercially sold smart phone, as they represent a common, yet widely varying platform to which many people have access. Because it is one of the most used open-source OS, Android was chosen. The Galaxy Nexus was used as test device.

Galaxy Nexus

Android OS: 4.2.2
Kernel: 3.0.31-g9f818de
A factory reset was executed at the beginning of the tests, providing a clean installation. Throughout most of the measurements the phone was running in airplane mode, with the exception of one case as mentioned in a later section. To reduce the amount of tasks running during our measurements

we used an application terminator, called *Advanced Task Killer*[1]. Previous to each measurement *Advanced Task Killer* was executed.

While most of the measurements were taken with the phone only, for some verification we used an oscilloscope and a multimeter. Due to the lack of a current meter with the ability to log measurements, we placed a resistance between the negative pole of the battery and the phone, and then logged the voltage-drop over the resistance. In a first attempt, we plugged a PCboard with a resistor between the phone and the battery. With the oscilloscope we measured the voltage over the battery and the voltage over the resistor. Choosing the smallest resistance (1 mΩ) we had available in the laboratory, we took our first measurements. However, the analysis showed that the noise in the measurements over the resistor was too big to allow useful conclusions to be drawn. Consequentially, in a next test, we used a bigger resistance (5 mΩ), but the signal to noise ratio was still too low. Also the repeated moving of the phone deformed the phone contacts that the board was connected to. In a second attempt, we glued two flat wires together, removing the insulation on the opposite sides. We were then able to plug in this wire-bridge between the phone and the battery while inserting the battery into the phone in a normal fashion, effectively removing the stress on the phone contacts. This wire-bridge is shown in Figure 3.1. Coming out of the wire-bride were three uninsulated wires, one for the plus pole of the battery, one for the negative pole of the battery and one for the phone contact, where the negative battery pole would be connected to. The open end of the negative battery pole and the phone contact were then bridged by the multimeter. Unfortunately, the multimeter was not capable of logging the measurements, so it served as a meter for sporadic current readings and as shunt-resistor over which we logged the voltage-drop with the oscilloscope. Thus we logged two different voltages with the oscilloscope, one over the battery and one over the multimeter.

For the evaluation we wrote two applications. The CodeSampler periodically measures, logs the data and plots the energy consumption of the device during an interval. The CodeTester executes the codes that we want to compare for their energy consumption and logs measurements after each code iteration. Both applications have a GUI, where a few settings can be adjusted, but their main duty is run in the background.

For evaluating the log files we used *MATLAB*[2].

[1] https://play.google.com/store/apps/details?id=com.rechild.advancedtaskkiller
[2] http://www.mathworks.com/products/matlab

Figure 3.1: Wire-Bridge
Construction to reduce the stress on the phone contacts

3.2 Evaluation

In this section we present various case studies we conducted.

3.2.1 Power Evaluation

In order to create discharge curves we need to be able to drain power at a constant rate. We chose to implement a number of different sorting algorithms for draining power, as we could later compare them against each other for their power consumption. For more information on the comparison of the sorting algorithms, please refer to Section 3.2.5. The validity of our method to drain a constant amount of power is evaluated by using an oscilloscope, the setup is as explained in the previous section. Figure 3.2 depicts the voltage measurement over the multimeter of a complete discharge. During the measurement we ran an Insertion Sort algorithm with an array of 100'000 integers. Besides having a few anomalies, no obvious trend of increasing or decreasing voltage could be observed.

To reduce the noise we took 500 samples and calculated their mean value,

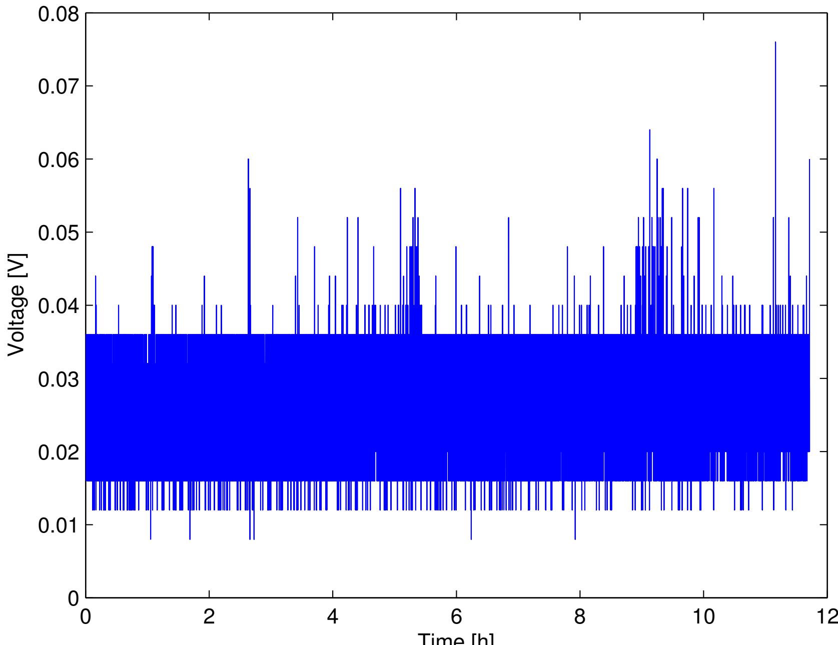


Figure 3.2: Voltage over Multimeter
The voltage measured over the multimeter with the oscilloscope during a full battery discharge

each value representing a new point in Figure 3.3. As can be seen in Figure 3.3, the voltage over the multimeter increases with time, meaning that the current increases during a discharge. This makes perfect sense as the voltage over the phone decreases and power (P) is the product of voltage (V) and current (I).

$$P = V_{phone} * I = V_{phone} * \frac{V_{meter}}{R_{meter}}$$

To verify that the power during the discharge is constant, we compared the values at the beginning and the end of the discharge. We took the average of the first and last 2'500 samples for this calculation, the samples were taken at a constant 2.5 samples per second:

$$P_{start} \approx P_{end}$$
$$V_{phone_{start}} * \frac{V_{meter_{start}}}{R_{meter}} \approx V_{phone_{end}} * \frac{V_{meter_{end}}}{R_{meter}}$$
$$4.085V * 0.0259V \approx 3.623V * 0.0278V$$
$$0.1057 \approx 0.1008$$

This resulted in a divergence of about 5% for the power values. The assumption that the power consumption is constant seems reasonable.

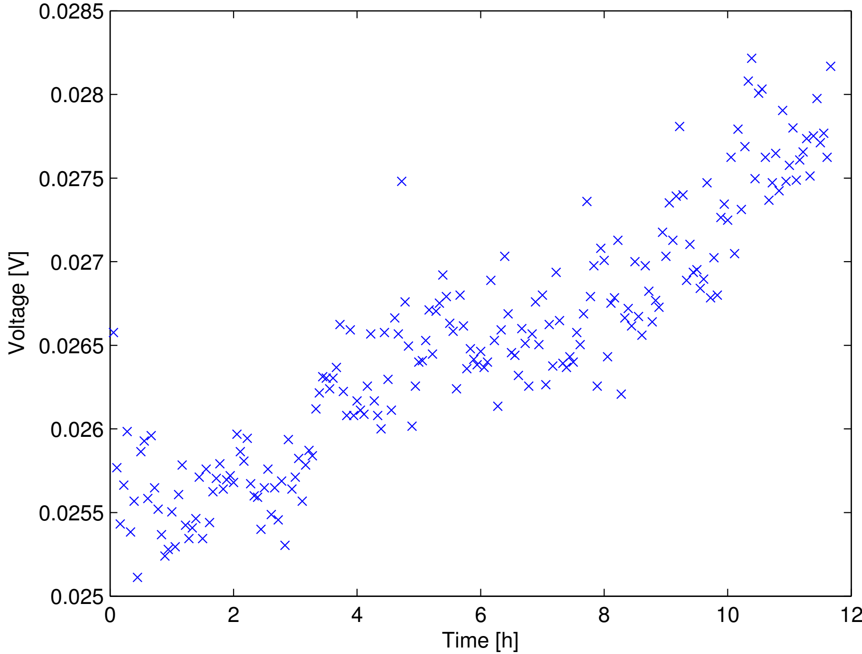


Figure 3.3: Voltage over Multimeter Averaged
The voltage measured with the samples averaged

3.2.2 Discharge Curve

Possessing a way to drain a constant amount of power, the next step is to verify our assumption that the discharge curves are monotonically falling. To achieve that we repeatedly executed sorting algorithms, while using our application to log the battery voltage.

Figure 3.4 shows the curve of a complete discharge, using the Quick Sort algorithm on a 1'500'000 integer array. We can see a lot of noise in the measurement. This noise in fact is bigger than the granularity of the internal voltage meter in the phone. This introduces an error we did not account for in the error analysis. To counter this additional error we used *MATLAB*'s smoothing function on the curves before creating our merged discharge curve, effectively removing any spiking voltage excursions. In Figure 3.5 our resulting merged discharge curve is depicted.

Note that we recommend to use the steep parts of the discharge curve for charge approximations. As we can see in Figure 3.6, when a voltage has

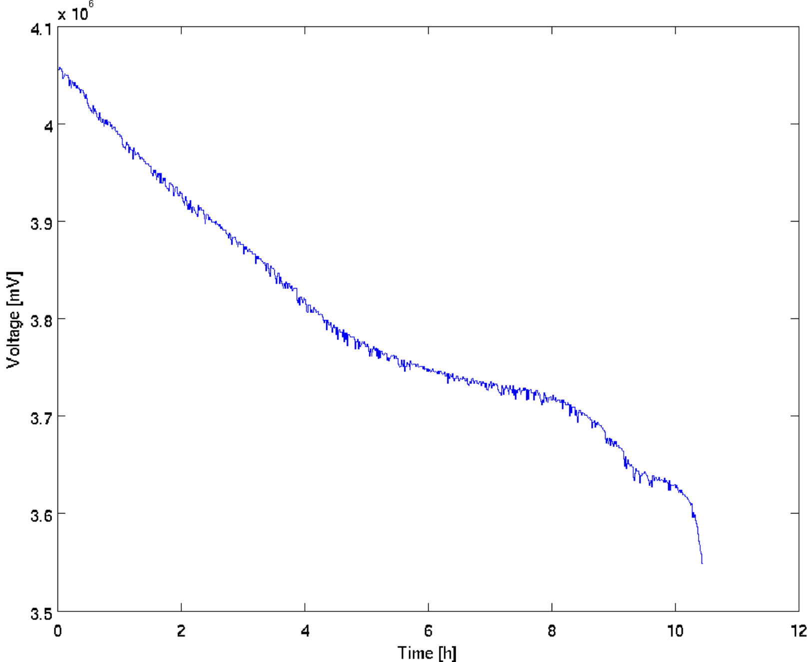


Figure 3.4: Discharge Curve
A complete discharge curve measured with the oscilloscope

a certain error range a flat discharge curve amplifies the error range of the estimated charge. In the case of our merged discharge curve it is advisable to use the first (steeper) part of the discharge curve.

3.2.3 Charge Estimation

This subsection focuses on comparing our charge estimation against the battery level provided by the Android OS. Also, we compare both approximations against the straight line that we would expect a constant power drain to produce. The OS values are read from the OS intent service *Intent.ACTION_BATTERY_CHANGED*. From this intent we can read the extra *BatteryManager.EXTRA_LEVEL* for the current battery level scaled by another extra *BatteryManager.EXTRA_SCALE*. The level values are an integer field and for our phone the scale is 100, leaving us with a step size of 1%.

Figure 3.7 shows the approximated charges for our method and the OS during a discharge of the Quick Sort algorithm on a 5'000'000 integer array. We can see that the region under 50% charge has more fluctuations, this is expected as the discharge curve is flat in this region and thus amplifies inaccuracy.

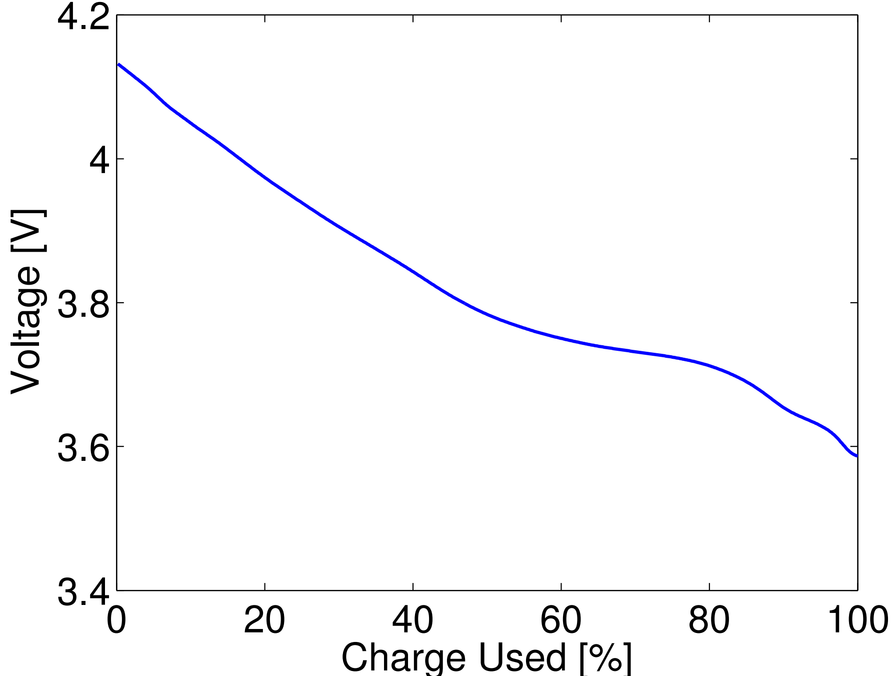


Figure 3.5: Merged Discharge Curve
Resulting merged discharge curve

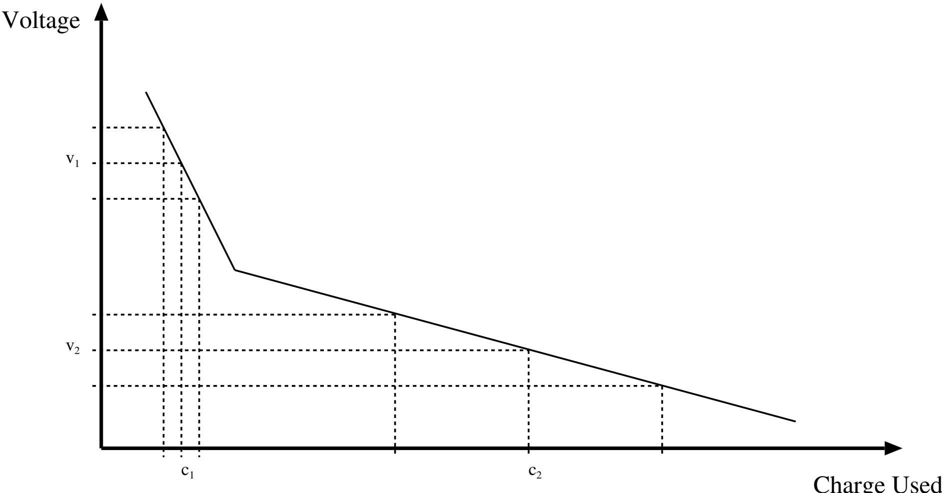


Figure 3.6: Effect of Discharge Curve Slope
Effect of the slope of the discharge curve on the range of the estimated charge

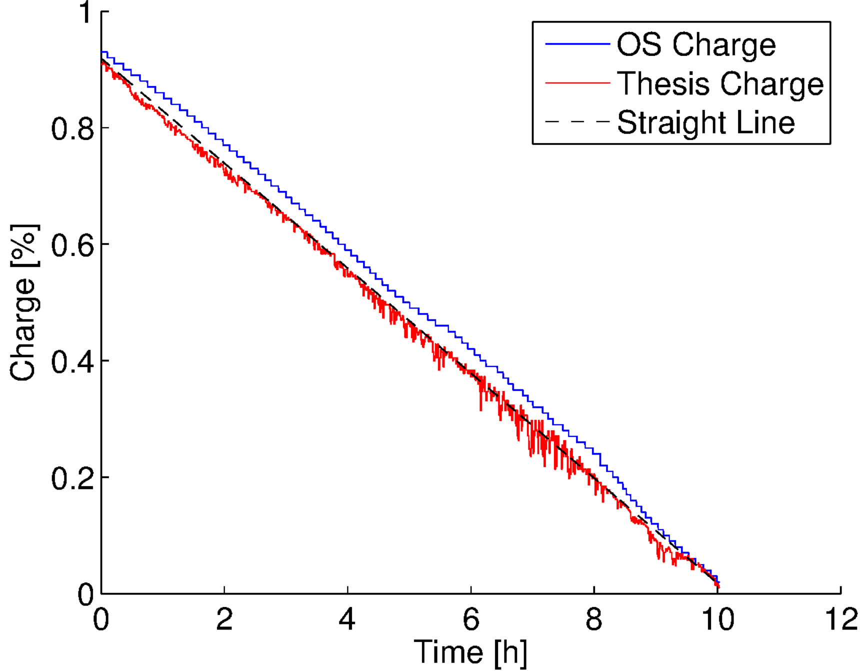


Figure 3.7: Charge Estimation
Charge estimation over a complete discharge

So we recommend to perform measurements above 50% charge for better accuracy.

3.2.4 Differentiating Components

We were also interested whether we would be able to differentiate hardware components in their energy consumption. Thus we logged the executing Insertion Sort on a 100'000 integer array with the screen of the Galaxy Nexus turned on. After a while we then stopped the execution of the code and only left the screen on. Figure 3.8 depicts the power consumption of this measurement.

The difference in their consumption is so big, that there is no overlapping of the error boundaries. Hence we can conclude that our method can easily and clearly distinguish the two mentioned cases.

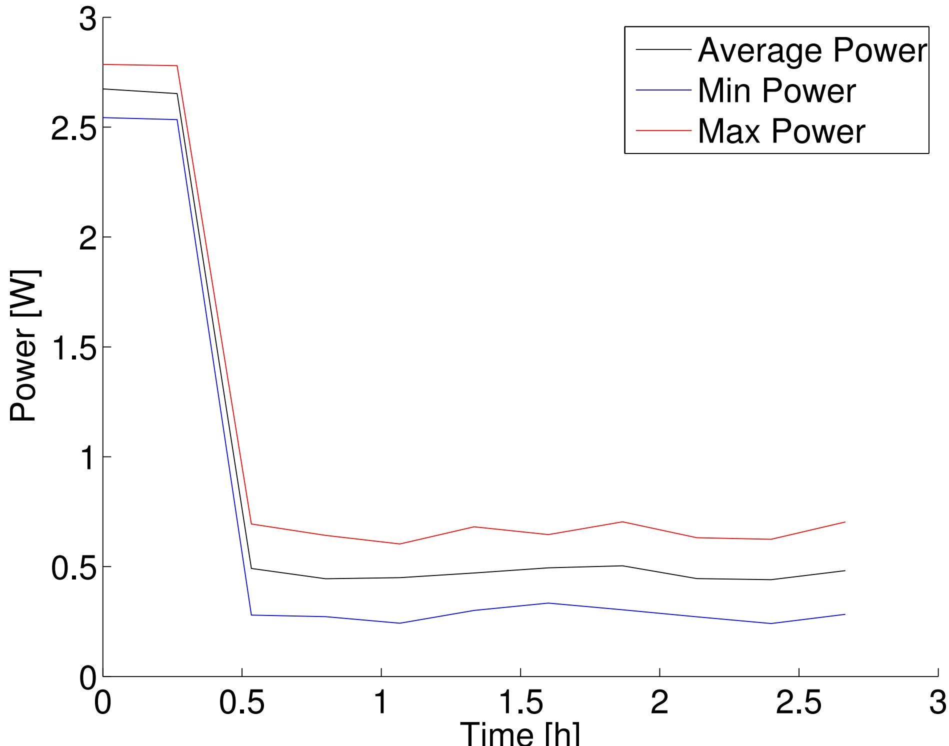


Figure 3.8: Insertion versus Screen
In the first part we have an Insertion Sort algorithm with turned on Screen, in the second part only the screen is active

3.2.5 Comparison of Different Sorting Algorithms

An important feature we wanted our application to have, was the ability to compare different ways of implementing a solution. For illustration, we chose sorting algorithms, as they are used in many programs. See Appendix A for a few examples of sorting algorithms. To create a comparison we picked the following four sorting algorithms:

- Bubble Sort
- Insertion Sort
- Quick Sort
- Merge Sort

The sorting algorithms were implemented in a Java Android application, see Appendix B for the source code of the algorithms. No multitasking was used in the implemented versions. Sorting algorithms can be applied to many

different types (bytes, integers, etc.), in the following examples we limited ourself to integer as the type to be sorted.

Standard Java random generator (java.util.Random) is used for populating the sorting array, the generator seed can be change in the settings of the application, the seed used was 123456.

Before and after a sorting algorithm has finished sorting an array, the voltages and the timestamps are logged.

After running each type of sorting algorithm, we observed that Merge Sort and Quick Sort are executed much faster than Bubble Sort and Insertion Sort. This was to be expected as their (timing) performance, executed on an array sized n, is $O(nlog(n))$ in comparison to Bubble Sort and Insertion Sort (n^2).

The respective execution times for applying the four sorting algorithms to a set of 150'000 integers were:

Name	Merge Sort	Quick Sort	Bubble Sort	Insertion Sort
Runtime[ms]	251.5	224	1'149'357	746'235

We see a significant difference between Merge Sort/Quick Sort and Bubble Sort/Insertion Sort. Note that the runtime of the Merge Sort and Quick Sort is far below the update interval of the voltage file (50s), which often results in having the same voltage at the beginning and ending of a sorting iteration. Thus we decided to increase the array sizes we use for those two algorithms.

Figure 3.9 depicts the energy consumption and the time used for sorting of different sized arrays by Bubble Sort and Insertion Sort. Merge Sort's and Quick Sort's energy consumption and time used are shown in Figure 3.10. More detailed information can be found in the Appendix C.

Between the different algorithms we have big variations in the energy consumption and runtime, however variations in the power consumption is small in comparison to those of the energy and runtime. This implies that for these sorting algorithms, the energy consumption seems to be almost proportional to the runtime. With increasing array size, as expected, the energy consumption and the runtime increases. Similar to the difference in runtime between Quick/Merge Sort and Bubble/Insertion Sort we mentioned earlier, we can observe a difference in energy consumption, as Quick/Merge Sort consume less energy even with the much bigger arrays. Quick Sort both runs the fastest and consumes the least energy per sorting iteration, followed in both categories by Merge Sort. This clearly leads to the conclusion, that if you have to implement an integer sorting algorithm on an Android device, out of the examined algorithms, Quick Sort would be the recommended one.

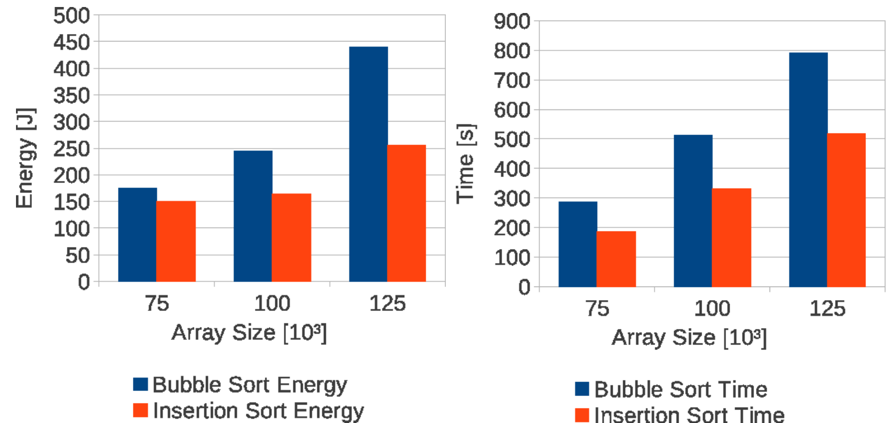


Figure 3.9: Bubble Sort and Insertion Sort Evaluation
On the left side we see the energy used by the Bubble Sort and Insertion Sort algorithms, the right side shows the time used

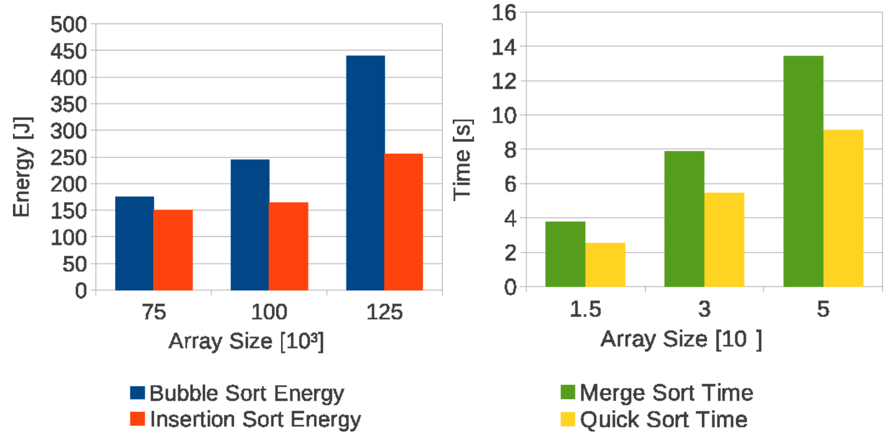


Figure 3.10: Merge Sort and Quick Sort Evaluation
On the left side we see the energy used by the Merge Sort and Quick Sort algorithms, the right side shows the time used

3.2.6 Evaluation of Common Applications

In this section we present a few examples of commonly used applications on the Android device. Figure 3.11 shows the different power consumptions of listening to music, watching a movie on Youtube[3] and playing Angrybirds[4]. We can see that even though the movie was streamed over WiFi, which is considered a big power consumer for mobile devices, Angrybirds still consumes more power. As expected listening to music consumed far less power

[3]https://www.youtube.com/watch?v=od3pZzaLP8A
[4]http://www.angrybirds.com/

than multimedia or gaming, as they both feature music as integral part.

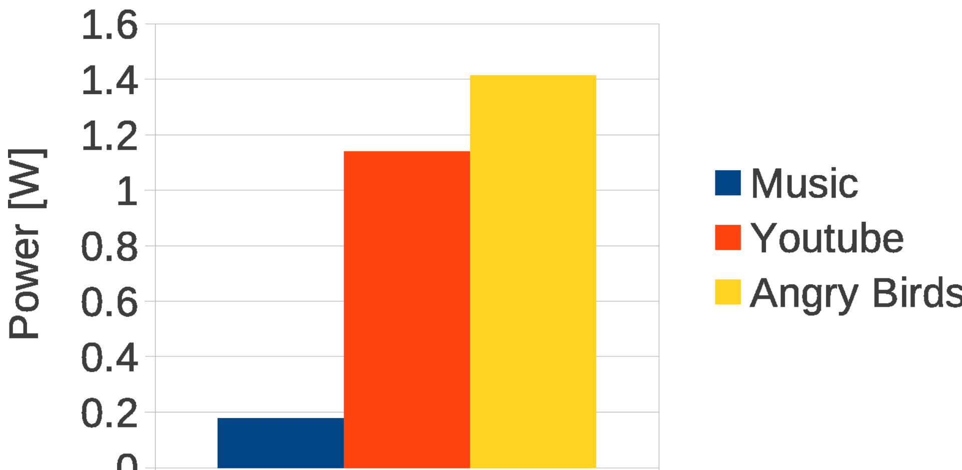


Figure 3.11: Comparison of Common Applications
Comparison of the power consumption of playing music, watching a movie on Youtube and playing Angrybirds

Youtube Movie over WiFi

We used the built-in Youtube application on our Galaxy Nexus to watch the movie *Last Man Standing*. In order to connect to the Internet we started the WiFi, so we had to deactivate the phone's airplane mode. Figure 3.12 depicts the voltage curve logged during the movie.

A steep drop of the voltage can be observed in the first part of the curve. At around 1.6 h the movie ended and the voltage no longer decreases as fast as before. The curve is not monotonically falling, over the entire curve we measured 15 voltage increases. This implies that energy predictions over short times can be very inaccurate, as these variations are rather large in comparison to the overall voltage drop. If the energy consumption is calculated over the whole movie the energy estimation is considerably more accurate and no negative energy is calculated. Thus in this case our method requires a big enough time interval for approximating the energy accurately, but fundamentally still works. The causes of these big voltage decreases and increases need to be further investigated. They possibly result from changes in the power draw from the battery, during high consumption periods.

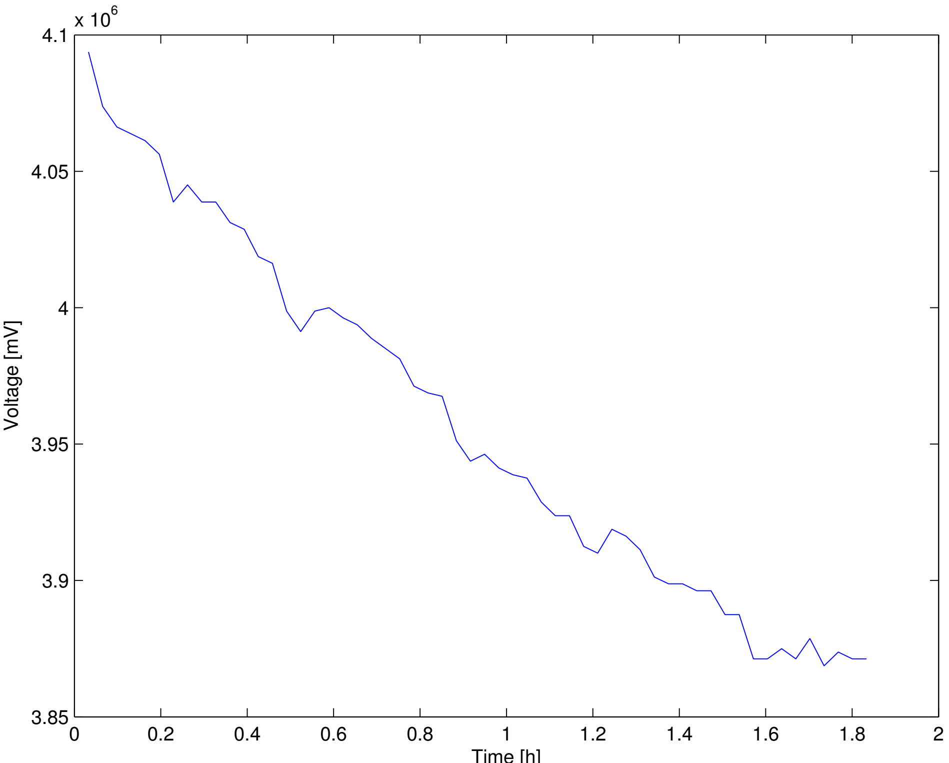


Figure 3.12: Youtube Movie Voltage
This figure shows the voltage curve during the playing of a movie on Youtube

4

Conclusion and Outlook

4.1 Conclusion

This master thesis proposed a software-based method to approximate energy consumption on battery-powered devices. The only inputs the method depends on are the battery capacity, the instantaneous voltage and, for power estimations, the time. Discharge curves play a pivotal role in our method, as they are used to convert the measured voltage to the relative charge state of the battery. Further, we derived error boundaries for the different stages of our energy approximation. After implementing our method on an Android device, we evaluated it in various case studies for its effectiveness. Looking at the approximated charge after a constant discharge cycle, we were able to see the expected linear decrease, suggesting the usefulness of our method. One important finding is that the steepness of the discharge curve affects the accuracy of the results, as flatter discharge curve regions amplify the inaccuracy of the battery charge approximation.

We have been able to make a contribution to the energy efficiency discussion, as we can show that our method is capable of differentiating various applications with regard to their energy consumption. In particular, we compared various sorting algorithms against each other and concluded which are the more energy efficient ones. Our evaluation of the sorting algorithms lead to the conclusion that from Bubble Sort, Insertion Sort, Merge Sort and Quick Sort, this last one is both the fastest and the most energy efficient algorithm.

4.2 Outlook

Our method depends heavily on the accuracy of the voltage measurements. Any further development to obtain more precise, frequent or consistent measurements would improve the overall accuracy of our approach. For example a deeper knowledge of how the voltage file is updated by the system driver might prove very useful.

While reviewing our logged voltages we observed cases where the battery voltage increased significantly without charging the battery. This might come from the battery's internal discharge behaviour under high power consumption. Further research into such behaviour could also improve the accuracy of the method.

We use the battery capacity written on the battery, but due to different influences (e.g. battery age) the effective capacity might change. An integrated estimation of the effective battery capacity could result in more accurate energy approximations.

Bibliography

[1] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, pp. 105–114, 2007.

[2] A. Ilka and F. Thomas, *Elementary Geometry.* AMS, Aug. 2008.

Examples of Sorting Algorithms

The following table was created with information found on *Wikipedia*[1];

[1] http://en.wikipedia.org/wiki/Sorting_algorithm

	Best case	Average case	Worst case	Allows parallelization	Description
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$	No	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
Quick sort	O(n logn)	O(n logn)	$O(n^2)$	Yes, the two sublists can be sorted separately	Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists. The steps are: 1. Pick an element, called a pivot, from the list. 2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation. 3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values. The base case of the recursion are lists of size zero or one, which never need to be sorted.
Merge sort	O(n log n)	O(n log n)	O(n log n)	Limited, sublists can be sorted separately but then need to be merged together	1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted). 2. Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	No	In each repetition this sorting algorithm removes one element from the unsorted input and adds it at the correct place in the sorted array.
Timsort	O(n)	O(n log n)	O(n log n)	No	Derived from merge and insertion sort.

					1. Divides data into subsets of size 32-64, and insert sorts those. 2. Then these subsets are merged together.
Heap sort	O(n log n)	O(n log n)	O(n log n)	No	1. Heap (tree structure) of the input data is made. 2. Repeatedly the largest element of the heap is added to the sorted array.
Smooth sort	n	O(n log n)	O(n log n)	No	Heap sort variant, instead of binary head it uses a Leonardo number[1] based heap.

Source: Wikipedia

1 http://en.wikipedia.org/wiki/Leonardo_numbers

B

Algorithm Code

Below you find the source code of the sorting algorithms used in the thesis is listed. Only the integer variant is shown, as the others simply replace the type (double, byte).

B.0.1 Support Functions

This section contains functions which are used by multiple sorting algorithms.

```
/**
 * swaps two values in an int array
 * @param intData
 * @param index1
 * @param index2
 */
final static void intSwap(int index1, int
    index2) {
        int tmp = intData[index1];
        intData[index1] = intData[index2];
        intData[index2] = tmp;
}
```

B.0.2 Bubble Sort

```
/**
```

```
 * uses bubble sort to sort data
 * (algorithm from wikipedia http://en.
    wikipedia.org/wiki/Bubble_sort)
 * @param intData
 */
static private void intBubbleSort() {
        int n = intData.length;
        do{
                int newn = 1;
                for (int i=0; i<n-1; ++i){
                        if (intData[i] >
                           intData[i+1]){
                                intSwap(i, i+1)
                                    ;
                                newn = i+1;
                        }
                }
                n = newn;
        } while (n > 1);
}
```

B.0.3 Insertion Sort

```
/**
 * insertion sort
 * algorithm from wikipedia http://en.wikipedia
    .org/wiki/Insertion_sort
 * @param intData
 */
static private void intInsertSort() {
        int insertValue;
        int currentPosition;
        for (int i = 0; i < intData.length; i
            ++){
                insertValue = intData[i];
                currentPosition = i;
                while (currentPosition > 0 &&
                   insertValue < intData[
                   currentPosition-1]) {
                        intSwap(currentPosition
                           , currentPosition -
                           1);
                        currentPosition--;
                }
```

```
        }
    }
```

B.0.4 Quick Sort

```
/**
     * from http://www.vogella.com/articles/
        JavaAlgorithmsQuicksort/article.html
     * Copyright © 2009 -2010 Lars Vogel
     * Eclipse Public License - v 1.0
     * @param low
     * @param high
     */
    static private void intQuicksort(int low, int
       high) {
            int i = low, j = high;
            // Get the pivot element from the
               middle of the list
            int pivot = intData[low + (high-low)
               /2];
            //TODO wiki mentiones different choices

            // Divide into two lists
            while (i <= j) {
                    // If the current value from
                       the left list is smaller
                       then the pivot
                    // element then get the next
                       element from the left list
                    while (intData[i] < pivot) {
                            i++;
                    }
                    // If the current value from
                       the right list is larger
                       then the pivot
                    // element then get the next
                       element from the right list
                    while (intData[j] > pivot) {
                            j--;
                    }

                    // If we have found a values in
                        the left list which is
                       larger then
```

```
                    // the pivot element and if we
                       have found a value in the
                       right list
                    // which is smaller then the
                       pivot element then we
                       exchange the
                    // values.
                    // As we are done we can
                       increase i and decrease j
                    if (i <= j) {
                            intSwap(i, j);
                            i++;
                            j--;
                    }
            }
            // Recursion
            if (low < j)
                    intQuicksort(low, j);
            if (i < high)
                    intQuicksort(i, high);
    }

\end{verbatim}

\subsection{Merge Sort}
%\begin{verbatim}
\begin{lstlisting}
/**
* from http://www.vogella.com/articles/
    JavaAlgorithmsQuicksort/article.html
* Copyright © 2009 -2010 Lars Vogel
* Eclipse Public License - v 1.0
* @param low
* @param high
*/
private static void intMergesort(int low, int high) {
        // check if low is smaller then high, if not
           then the array is sorted
        if (low < high) {
                // Get the index of the element which
                   is in the middle
                int middle = low + (high - low) / 2;
                // Sort the left side of the array
                intMergesort(low, middle);
```

```java
                // Sort the right side of the array
                intMergesort(middle + 1, high);
                // Combine them both
                intMerge(low, middle, high);
        }
}

/**
* from http://www.vogella.com/articles/
    JavaAlgorithmsQuicksort/article.html
* Copyright © 2009 -2010 Lars Vogel
* Eclipse Public License - v 1.0
* @param low
* @param middle
* @param high
*/
private static void intMerge(int low, int middle, int
    high) {

        // Copy both parts into the helper array
        for (int i = low; i <= high; i++) {
                intHelper[i] = intData[i];
        }

        int i = low;
        int j = middle + 1;
        int k = low;
        // Copy the smallest values from either the
            left or the right side back
        // to the original array
        while (i <= middle && j <= high) {
                if (intHelper[i] <= intHelper[j]) {
                        intData[k] = intHelper[i];
                        i++;
                } else {
                        intData[k] = intHelper[j];
                        j++;
                }
                k++;
        }
        // Copy the rest of the left side of the array
            into the target array
        while (i <= middle) {
                intData[k] = intHelper[i];
```

```
                k++;
                i++;
        }

}
```

Data Tables for Sorting Algorithms

Below are the tables with the measurements for the comparison of the four sorting algorithms. We use the following notation:
(x*y) after the algorithm name means that the estimation was made over x iterations and y times. The single values are the averaged values for a single algorithm execution, followed by the standard derivation in the brackets.

Bubble Sort and Insertion sorting is done over an array with size 75'000:

Name	Bubble Sort (5*9)	Insertion Sort (5*10)
Energy[mJ]	1.746083e+05 (7.441121e+04)	1.501189e+05 (6.001624e+04)
MinEnergy[mJ]	1.485683e+05 (7.663911e+04)	1.192043e+05 (6.160573e+04)
MaxEnergy[mJ]	2.044124e+05 (7.586935e+04)	1.812223e+05 (6.203937e+04)
Runtime[ms]	2.881365e+05 (1.986272e+02)	1.858828e+05 (4.480509e+02)
Power[W]	6.059449e-01 (2.581284e-01)	8.077267e-01 (3.237800e-01)

Bubble Sort and Insertion sorting is done over an array with size 100'000:

Name	Bubble Sort (5*6)	Insertion Sort (5*10)
Energy[mJ]	2.445118e+05 (1.693586e+04)	1.639335e+05 (1.678342e+04)
MinEnergy[mJ]	2.188978e+05 (1.839798e+04)	1.356985e+05 (1.796839e+04)
MaxEnergy[mJ]	2.727432e+05 (1.810825e+04)	1.907434e+05 (1.604059e+04)
Runtime[ms]	5.135351e+05 (1.787054e+02)	3.314993e+05 (3.587402e+02)
Power[W]	4.761408e-01 (3.307094e-02)	4.945610e-01 (5.101710e-02)

Bubble Sort and Insertion sorting is done over an array with size 125'000:

Name	Bubble Sort (5*3)	Insertion Sort (5*5)
Energy[mJ]	4.407592e+05 (1.390886e+05)	2.562188e+05 (2.008448e+04)
MinEnergy[mJ]	4.015737e+05 (1.420630e+05)	2.216062e+05 (2.394022e+04)
MaxEnergy[mJ]	4.747745e+05 (1.296781e+05)	2.926452e+05 (1.931217e+04)
Time[ms]	7.922645e+05 (9.472554e+02)	5.176117e+05 (9.971855e+02)
Power[W]	5.562410e-01 (1.751759e-01)	4.950718e-01 (3.974052e-02)

Merge Sort and Quick Sort sorting an 1'500'000 integer array:

Name	Merge Sort (50*61)	Quick Sort (50*74)
Energy[mJ]	3.484381e+03 (1.898158e+03)	2.923804e+03 (1.877250e+03)
MinEnergy[mJ]	4.775286e+02 (2.135912e+03)	2.500141e+01 (1.964343e+03)
MaxEnergy[mJ]	6.454100e+03 (2.063662e+03)	5.839028e+03 (2.082196e+03)
Time[ms]	3.757775e+03 (1.732961e+00)	2.542361e+03 (1.435738e+00)
Power[W]	9.272826e-01 (5.052609e-01)	1.150122e+00 (7.385461e-01)

Merge Sort and Quick Sort sorting an 3'000'000 integer array:

Name	Merge Sort (50*29)	Quick Sort (50*32)
Energy[mJ]	6.206304e+03 (3.529190e+03)	3.554989e+03 (2.025836e+03)
MinEnergy[mJ]	3.102067e+03 (3.605517e+03	5.317715e+02 (2.194348e+03)
MaxEnergy[mJ]	9.288555e+03 (3.881317e+03)	6.555087e+03 (2.452027e+03
Time[ms]	7.860513e+03 (3.510509e+00)	5.440458e+03 (3.392596e+00)
Power[W]	7.895626e-01 (4.489304e-01)	6.533858e-01 (3.720311e-01)

Merge Sort and Quick Sort sorting an 5'000'000 integer array:

Name	Merge Sort (50*18)	Quick Sort (50*26)
Energy[mJ]	9.494990e+03 (3.176252e+03)	6.068519e+03 (2.500759e+03)
MinEnergy[mJ]	6.647479e+03 (3.308081e+03)	3.030633e+03 (2.716858e+03)
MaxEnergy[mJ]	1.224803e+04 (3.525544e+03)	9.104578e+03 (2.519000e+03)
Time[ms]	1.343516e+04 (4.175853e+01)	9.092024e+03 (8.875988e+00)
Power[W]	7.067369e-01 (2.362242e-01)	6.674883e-01 (2.750918e-01)

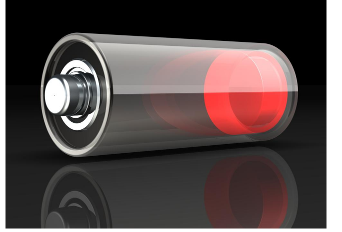
D
Presentation Slides

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Powerful Software

Master Thesis by Etienne Geiser
Advisers: Pratyush Kumar and Lars Schor

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Energy Efficiency

http://www.geek.com/wp-content/uploads/2012/06/low-battery.jpg
http://coloradopeakpolitics.com/wp-content/uploads/2013/08/Money-II-300x270.jpg
http://www.freegreatpicture.com/files/39/1609-tree.jpg

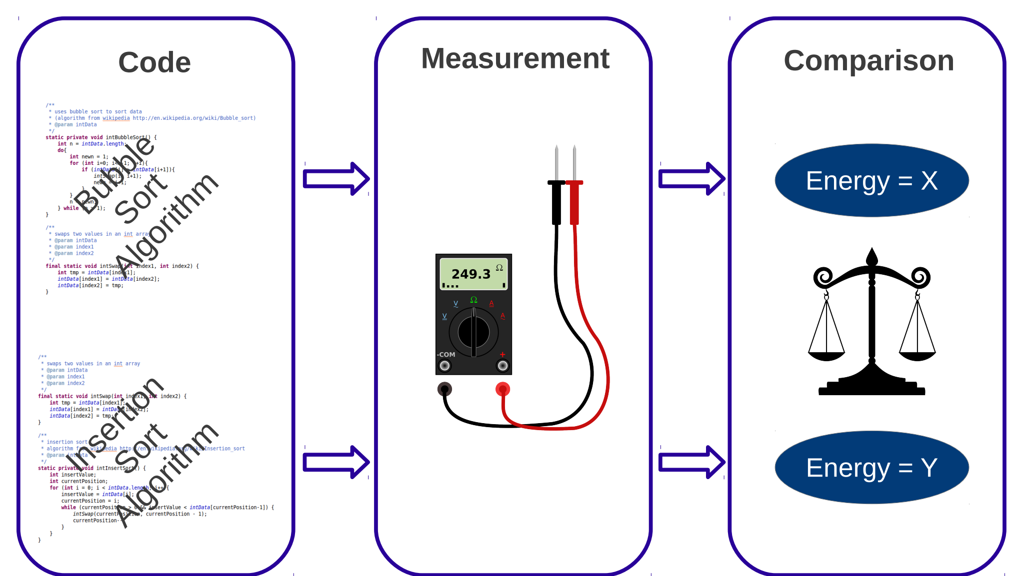
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Overview

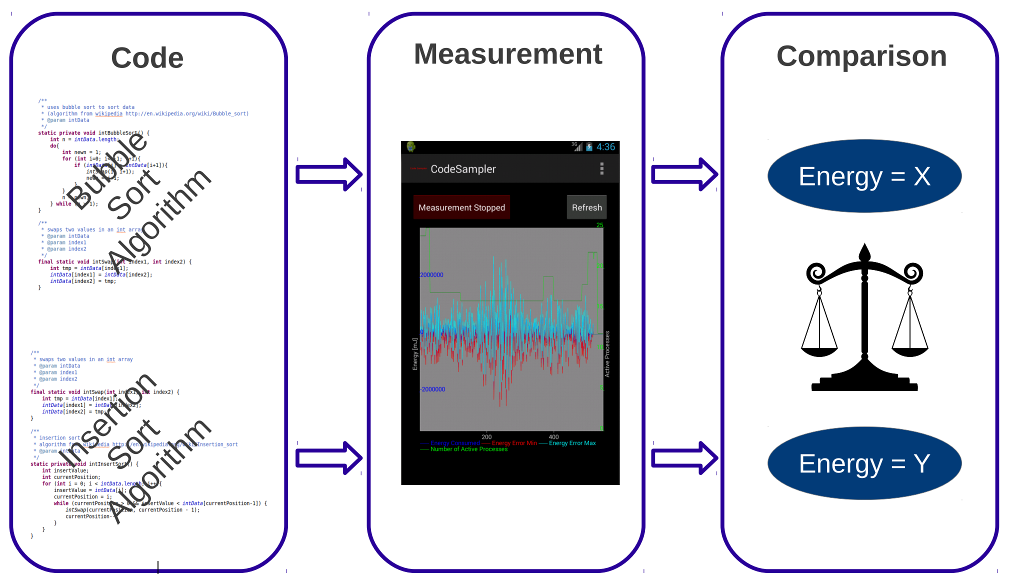
- Goal and Contribution
- Related Work
- Approach
  - Energy Calculation
  - Error Analysis
  - Implementation
- Evaluation
- Conclusion

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Goal



Goal



Contribution of this Thesis

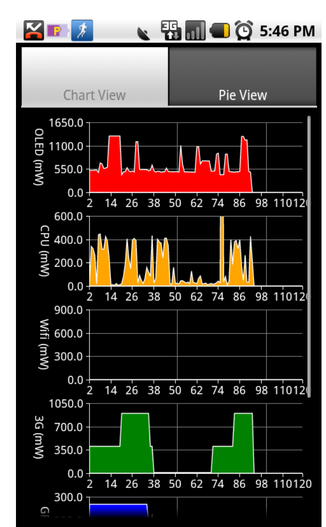
- We propose a software-based method to measure the energy consumption of battery-powered devices
- We derive error boundaries for energy consumption addressing inaccurate measurements
- We implement this method on an Android device and evaluate the effectiveness in various case studies

Overview

- Goal and Contribution
- **Related Work**
- Approach
  - Energy Calculation
  - Error Analysis
  - Implementation
- Evaluation
- Conclusion

PowerTutor

- Developed at the University of Michigan
- Existing power monitor for Android-based mobile platforms
- Issue:
  Knowledge of device components and their specific coefficients required

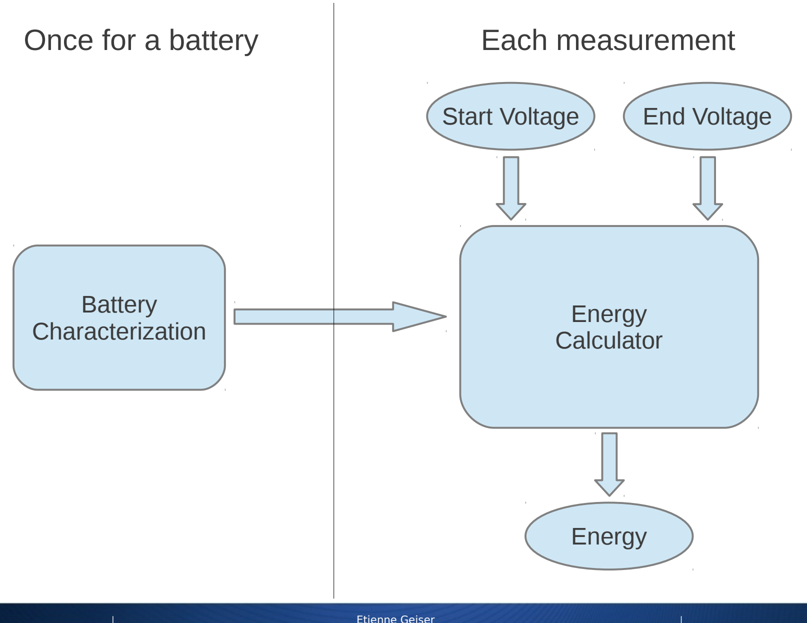


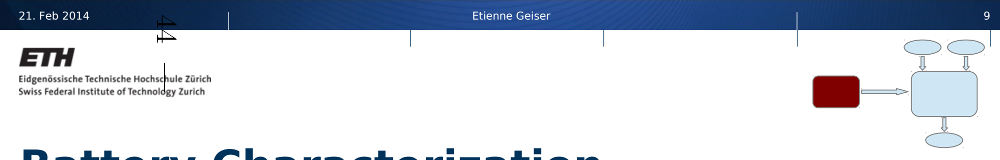
http://ziyang.eecs.umich.edu/projects/powertutor/

Overview

- Goal and Contribution
- Related Work
- **Approach**
  - Energy Calculation
  - Error Analysis
  - Implementation
- Evaluation
- Conclusion

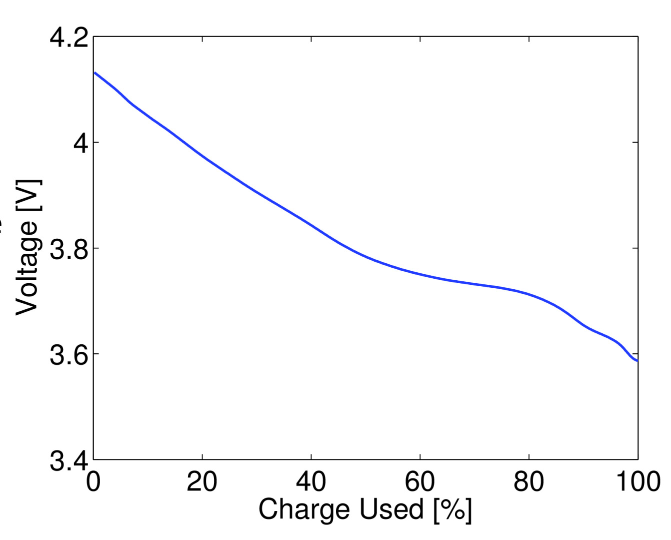
Approach



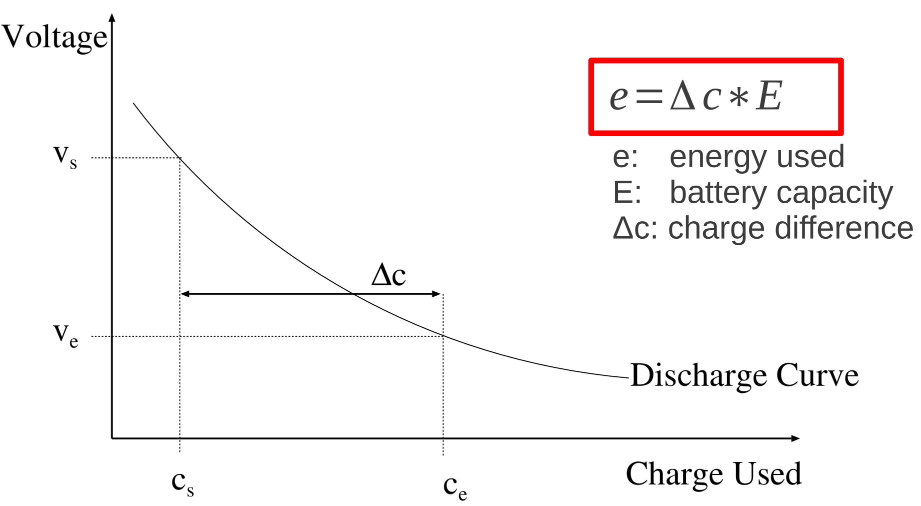


Battery Characterization

- Battery capacity
- Discharge curve
  - Battery voltage vs. battery charge
  - Drain full battery at constant rate



Energy Calculator



$$e = \Delta c * E$$

e: energy used
E: battery capacity
Δc: charge difference

Error Sources

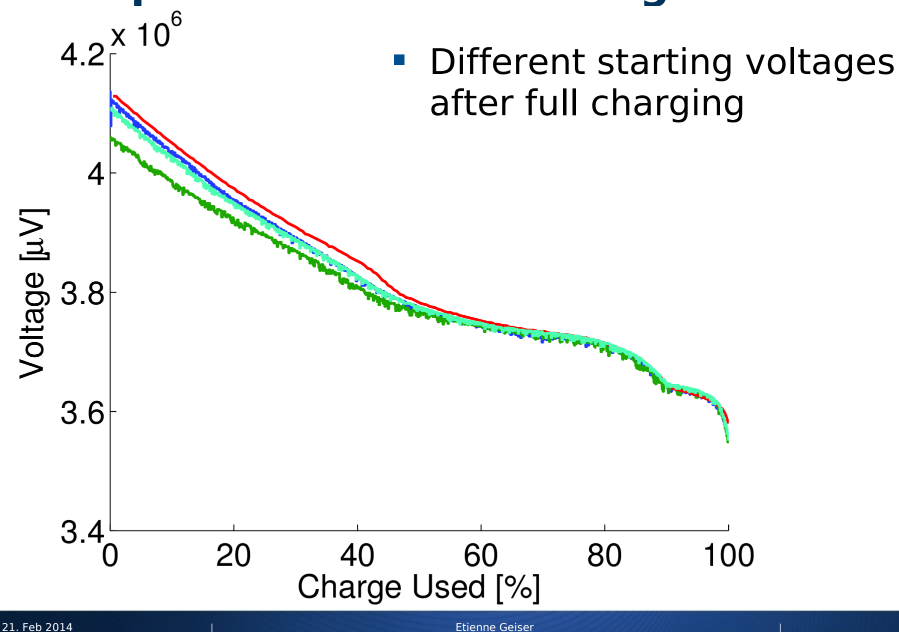
Inaccuracy of voltage measurements

Inaccuracy of voltage to charge conversion

Battery characteristics

- Ageing of battery leads to diverging discharge curves over time
- Environment affects discharge behaviour (e.g. temperature)

Examples of "Full" Discharges

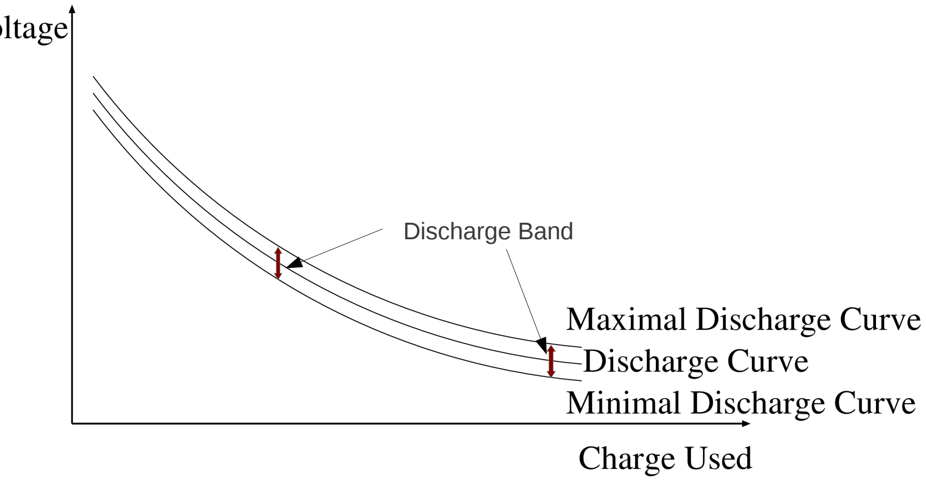


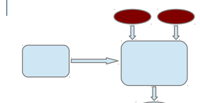
- Different starting voltages after full charging

Merged Discharge Curve

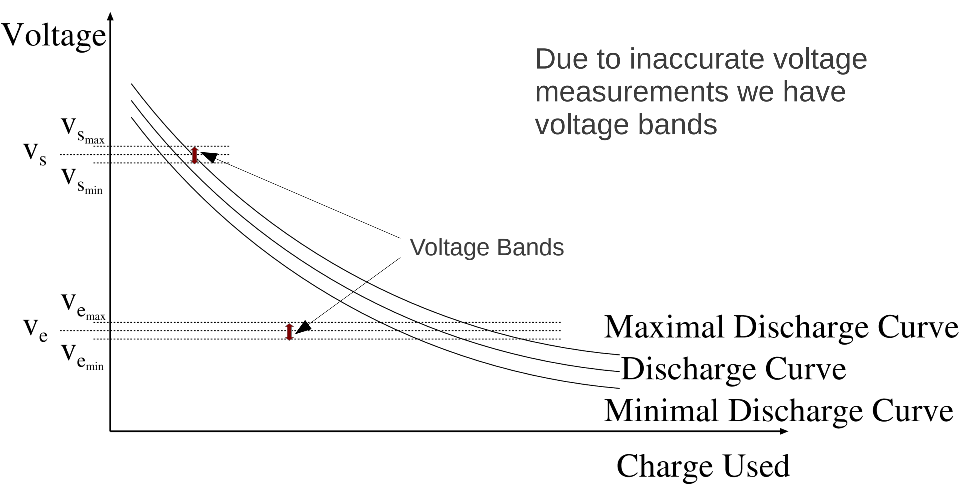
- Multiple curves are standardized and then averaged
- In order to be able to give an error bound we create three merged discharge curves:
  - Upper bound: Maximal merged discharge curve
  - Average merged discharge curve
  - Lower bound: Minimal merged discharge curve

Energy Approximation with Inaccuracies

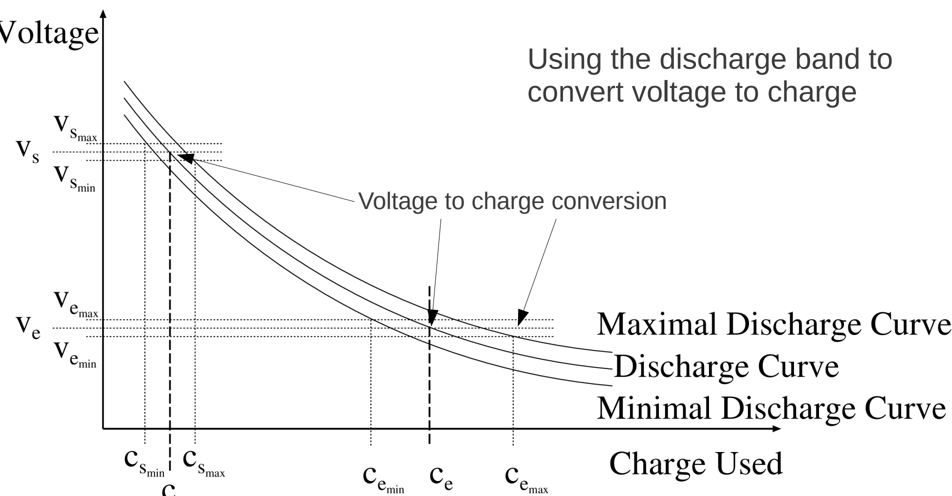


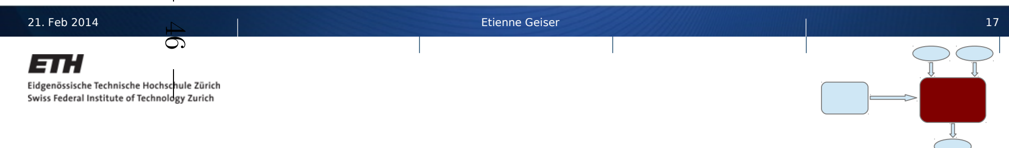


Energy Approximation with Inaccuracies

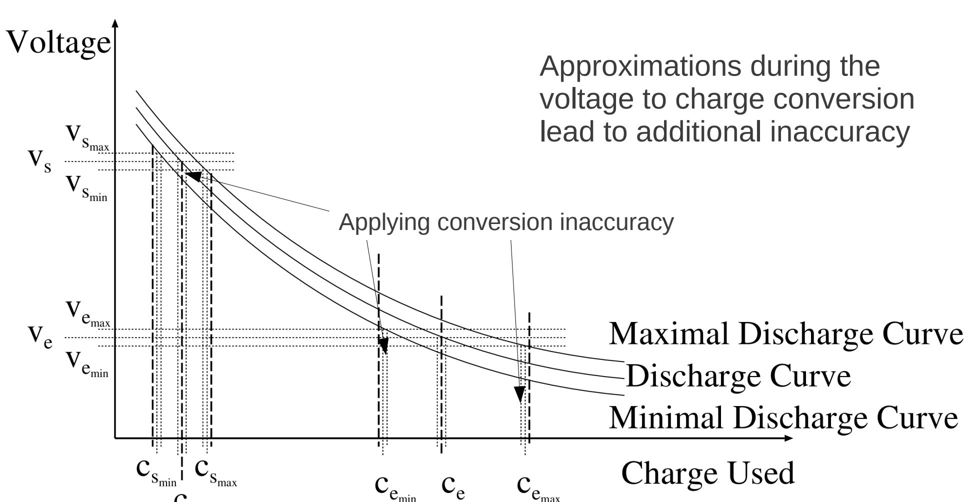


Energy Approximation with Inaccuracies

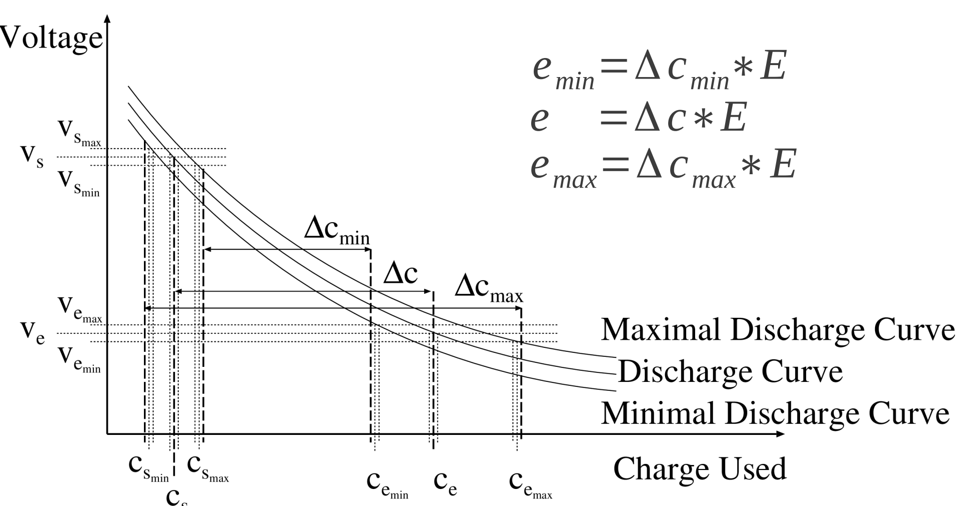




Energy Approximation with Inaccuracies



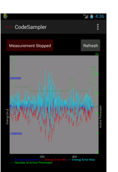
Energy Approximation with Inaccuracies



Implementation

- Android application for evaluation implemented
- Standard Android device: *Galaxy Nexus*

- *CodeSampler* application properties
  - Runs in background
  - Periodically reads measurements
  - Plots energy consumption graph on phone
  - Writes measured values into CSV file



- *MATLAB* used for in-depth evaluation

http://ameisenberg.com/wp-content/uploads/2012/12/samsung-android.jpg
http://www.mathworks.com/products/matlab

Overview

- Goal and Contribution
- Related Work
- Approach
  - Energy Calculation
  - Error Analysis
  - Implementation
- **Evaluation**
- Conclusion

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

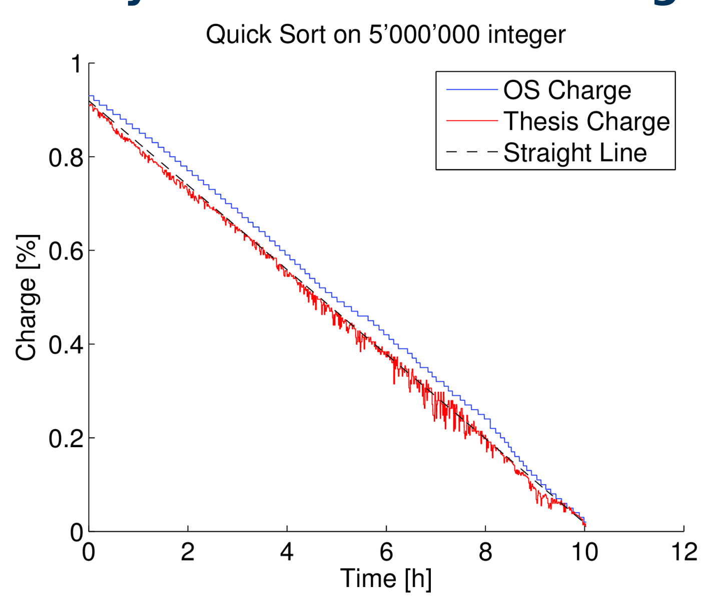
Evaluation Setup

- Factory reset
- Flight-mode active
- Running applications stopped with *Advanced Task Killer*
- Measurements taken at room temperature

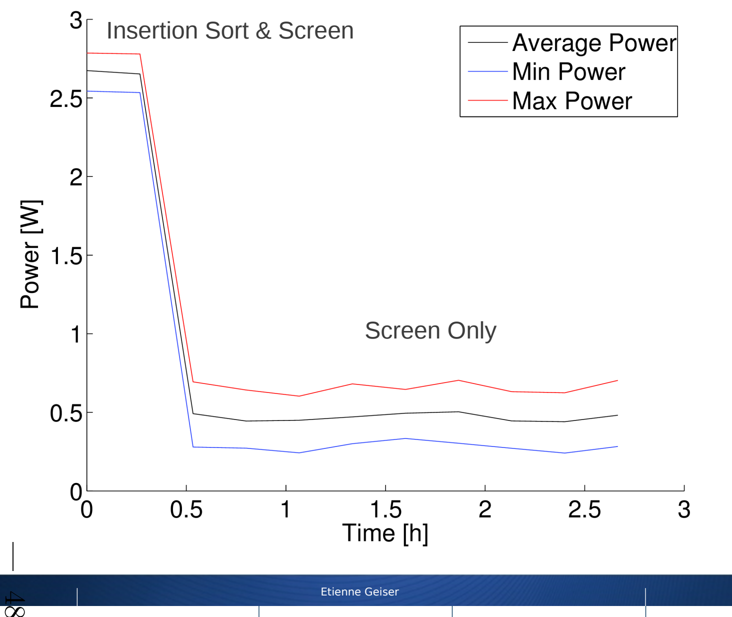
http://rechild.mobi/

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Case Study 1: Thesis vs OS Charge

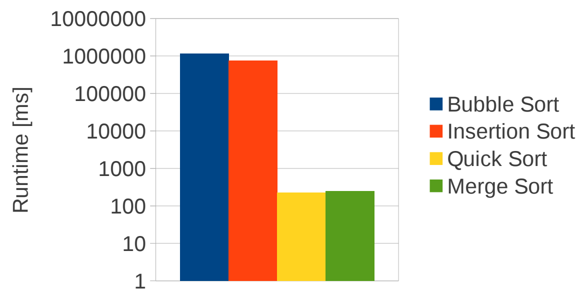


Case Study 2: Different Components

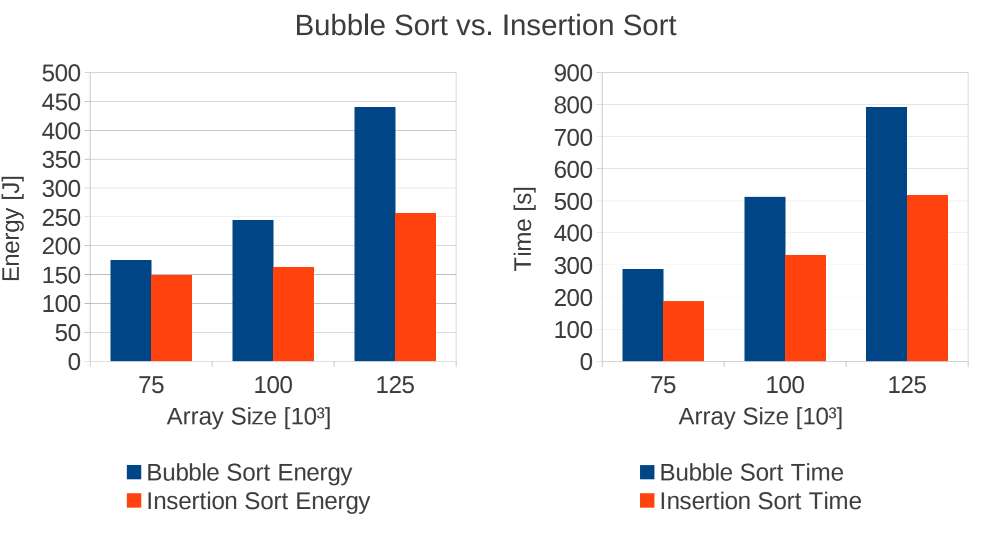


Case Study 3: Algorithm Comparison

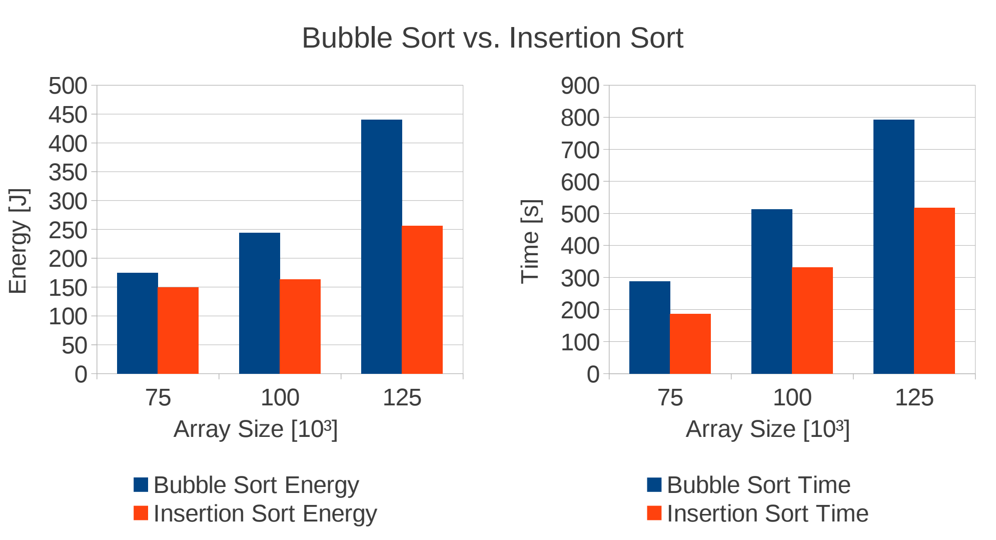
- Sorting algorithms used
  - Bubble Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort
- Runtimes on an integer array of 150'000 elements



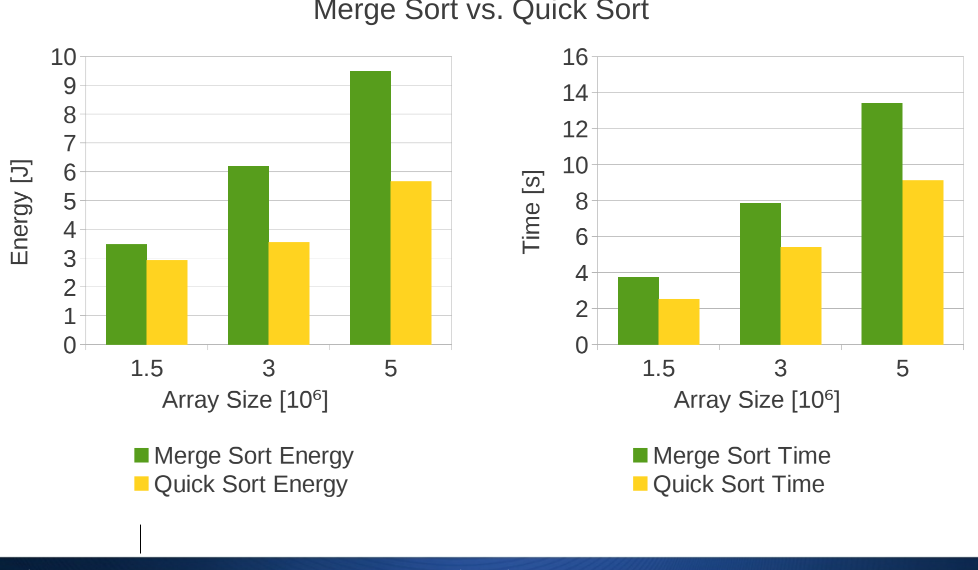
Case Study 3: Algorithm Comparison



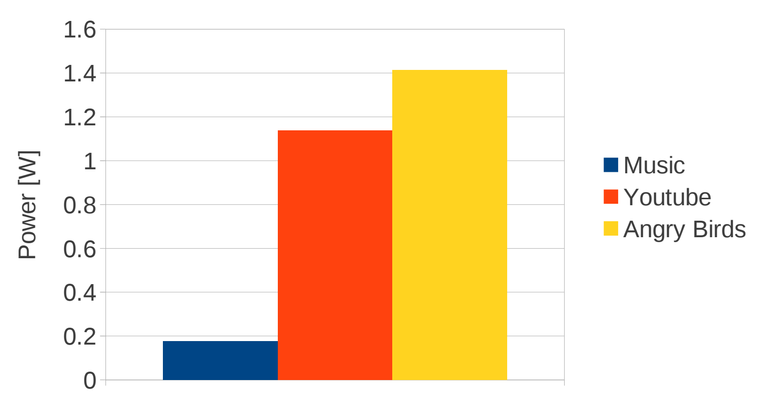
Case Study 3: Algorithm Comparison



Case Study 3: Algorithm Comparison



Case Study 4: Common Applications

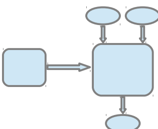


https://www.youtube.com/watch?v=od3pZzaLP8A
http://www.angrybirds.com/

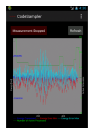
Conclusion

Contribution to energy efficiency discussion by:

- Developing a software-based power estimation method



- Implementing on Android



- Evaluating the usefulness in various cases