Aalborg University

Energy Measurement and Optimization of Continuous Gesture Recognition

Master Thesis in Computer Science

Author:
Jens Emil Gydesen

Supervisor:
Brian Nielsen

des103f16
February 2016 – June 2016

This page was intentionally left blank.

Department of Computer Science
Aalborg University

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Energy Measurement and Optimization of Continuous Gesture Recognition

Theme of project:
Master thesis in computer science

Project period:
February 2016 – June 2016

Participants:
Jens Emil Gydesen
(jgydes11@student.aau.dk)

Supervisor:
Brian Nielsen
(bnielsen@cs.aau.dk)

Abstract:
Energy consumption of software is becoming increasingly important as wearables, such as fitness trackers and other small battery-powered devices, are becoming a large part of our lives. These devices are typically controlled by a low-power CPU and powered by a small battery, and are thus very resource constrained. This thesis proposes a simple yet accurate method for developers to measure the energy consumption of their software. We demonstrate this method by implementing two continuous gesture recognition algorithms, and measuring their energy consumption. Furthermore we propose optimizations for these algorithms and gain up to 22 % energy reduction with no loss of accuracy, and up to 92.36 % energy reduction with 11 % loss of accuracy.

This page was intentionally left blank.

Energy Measurement and Optimization of Continuous Gesture Recognition

Jens Emil Gydesen
Department of Computer Science
Aalborg University
9220 Aalborg Øst, Denmark

Abstract

Energy consumption of software is becoming increasingly important as wearables, such as fitness trackers and other small battery-powered devices, are becoming a large part of our lives. These devices are typically controlled by a low-power CPU and powered by a small battery, and are thus very resource constrained. This thesis proposes a simple yet accurate method for developers to measure the energy consumption of their software. We demonstrate this method by implementing two continuous gesture recognition algorithms, and measuring their energy consumption. Furthermore we propose optimizations for these algorithms and gain up to 22 % energy reduction with no loss of accuracy, and up to 92.36 % energy reduction with 11 % loss of accuracy.

1 Introduction

Gesture based control, *i.e.* controlling devices through gestures, is becoming increasingly relevant, as more wearable devices are being developed and used [5]. Gesture recognition is already widely used today in pedometers, fitness and sleep trackers, as well as a range of smartphone, tablet and smartwatch applications. With smart homes also trending [4], we may see a increasing use of controlling your home with gestures by performing gestures using a wearable. These gestures are typically based on accelerometer, and sometimes gyroscope, sensor data, measured from the sensors in the wearable device. The sensors found in wearable devices today are typically three dimensional (3D), *i.e.* they gather data from the x, y and z axes.

Numerous solutions for (2D and 3D) gesture recognition [38, 39, 19, 14] and continuous gesture recognition [27, 42, 22] have been proposed, but only few [28, 40, 10] take energy as a resource constraint into account. These solutions typically try to optimize the *accuracy* of their gesture recognition algorithms, but as the devices doing the gesture recognition are often battery powered (*e.g.* smartphones, wearables or other embedded devices), recognizing the gesture may drain the battery needlessly.

Optimizing energy consumption of software running on battery powered devices can be very useful. Benini *et al.* [3] have found that *source code* optimization can reduce the energy consumption of up to 90 % and Šimunić *et al.* [32] likewise found up to 77 % energy reduction. The reason why we choose to measure and optimize for energy consumption, rather than runtime, is that for battery powered devices, battery life is a bigger concern than the runtime performance of the software. This is especially true for wearables such as fitness trackers, where there is a minimal interaction with the

software running, and thus less need for fast computations as they happen in the background. While reducing runtime typically also reduces energy consumption, measuring runtime does not take energy consumption of *e.g.* the DRAM into account, and runtime optimization techniques which utilizes DRAM for reduced runtime, may result in less energy consumption optimization, *i.e.* reducing runtime by 20 % does not necessarily mean 20 % energy reduction. Lastly, measuring energy consumption makes it possible to calculate the effect on the battery life of the device.
We setup the following research question:

> *How can we accurately measure the energy consumption of software, and measure the differences between optimized and unoptimized algorithms?*

While answering this, we will answer the following subquestions:

1. Which methods for measuring energy consumption exists?
2. Which of these methods are simple to use, and how accurate are their measurements?
3. Can we measure the effect of optimizing continuous gesture recognition algorithms?
4. Which algorithmic or architectural features contribute to the energy savings, *e.g.* fewer cache-misses, CPU instructions, *etc.*?

By answering these, we propose a simple method for measuring the energy consumption of the gesture recognition algorithms on battery powered devices. This method is directed at developers who want to measure energy consumption of their software. We implement and analyze two continuous gesture recognition algorithms, and present modifications to the algorithms which reduce their energy consumption. We evaluate the energy consumption optimization by measuring the energy consumption before and after the modification, while also comparing the accuracy and the aforementioned features.

The main contributions of this thesis is thus our proposed energy measurement method, and an analysis of which features are the main contributors to the energy consumption of the implemented gesture recognizers.

The remainder of this thesis is organized as follows. Section 2 describes related work to energy measurement methods and energy-optimized gesture recognition. As we cannot go into detail with all of the gesture recognition algorithms, we will present a pipeline of how continuous gesture recognition is performed, and briefly analyze and compare two gesture recognition algorithms in Section 3. We introduce energy optimization techniques and practices in Section 4, and present our method for measuring energy consumption in Section 5. We present the results of measuring the energy consumption and optimizing the algorithms in Section 6, and analyze which architectural or algorithmic features that contribute to the energy savings. We conclude the thesis and discuss the results, the method and further work in Section 7.

2 Related Work

Energy measurement and estimation for software has been researched extensively in the last two decades. Several different approaches have been used, each with advantages and disadvantages. These approaches can be divided into external energy measurements, *i.e.* energy measurements using *external* tools and sensors, and internal energy measurements, *i.e.* energy measurements using *internal* tools and sensors.

2.1 External Energy Measurements

The most common approach we found was instruction level analysis, which measures the energy of each instruction, and then uses these measurements to *estimate* the energy consumption. The reason why this is just an estimate, and not an accurate measurement, is that instructions *e.g.* consume different amounts of energy depending on which registers are being used and the data types being used *etc.* [26, 20].

This method requires external testing hardware (such as power meters) connected to the measured device. Many previous projects [33, 21, 26, 25, 13, 17, 10] have used this approach, where [21] managed to get up to 97.5 % accuracy compared to measurements from the external hardware tools. These approaches share the same method:

1. Use hardware measurement tools, such as power meters, to measure the average energy consumption of each instruction.
2. Use the result of the measurements to calculate the energy consumption of a program, by analyzing the instructions executed when running the program and summing the energy consumption of each instruction.

We will not go into details of each of the projects' equations for estimating the energy consumption, as they are mostly similar.

While this method can be accurate, and easy to use once the energy consumption of each instruction has been measured, instruction level energy consumption is almost impossible to calculate correctly, as the instructions' energy consumption vary depending on *e.g.* if the values are floating point numbers, which registers are being used, *etc.* [26, 20]. The downside of this method is that it requires measurements of the instructions beforehand. These numbers are typically not published by processor manufacturers like Intel, as it is a competition parameter. In order to perform this type of energy consumption analysis, you will thus have to rely on third party data or perform the measurements yourself, which is a long and expensive process. Furthermore, these measurements will also be very hardware/architectural dependent.

Unless such measurements are available, the instruction level analysis is an unattractive choice for developers.

2.2 Internal Energy Measurements

A number of different tools or framework that rely on *internal sensors*, such as the battery level and CPU cycle counter, have been proposed. These tools are easier to use as they do not need extra hardware, but may be limited in accuracy compared to the external measurements.

Alexander Bakker [1] has analyzed a number of different energy profilers for the Android OS. The profilers in [1] are mostly Android applications that can be used to measure the total energy consumption of an Android smartphone, or in some cases, the energy consumption of an application. EACOF [8] is a framework that utilize the battery level and CPU counters to determine the energy consumption of programs run on Linux or Mac OS. It enables programmers to measure the energy consumption of their software using an API. JouleUnit [37] is another framework that has can be integrated into the Eclipse IDE, providing the programmer an easy way to measure the energy consumption of his code. While these tools are easy to use, compared to the external hardware methods, they do not disclose the accuracy of their methods, and may be significantly less accurate. The tools which are based on the battery usage they also

measure energy usage of *e.g.* the computer screen, making it harder to get accurate application energy consumption.

Hähnel *et al.* [12] use the Running Average Power Limit (RAPL) sensors that are built into Sandy Bridge or newer Intel processors. They found RAPL to be very accurate compared to hardware measurements (only 1.13 % error rate on average), but their results showed an offset they attribute to DRAM energy consumption not being monitored in their tests, which were run on a CPU with a Sandy Bridge architecture. In Haswell and never architectures, the RAPL interface now also supports DRAM energy consumption. See Appendix E for a list of Intel CPU architectures that support RAPL.

2.3 Continuous Gesture Recognition Energy Optimization

We have only found three projects doing energy consumption optimization of continuous gesture, or activity, recognition. Ghasemzadeh *et al.* [10] use instruction level analysis of energy, and use the measurements to estimate the energy cost of classification features in activity recognition. By doing so, they can prioritize the features, *e.g.* the accelerometer on respectively the x, y, and z axes, which has low energy cost but yields high accuracy, effectively being able to remove less important sensors from a *sensor network*. Raffa *et al.* [28] does not perform any energy analysis, but utilize efficient algorithms to reduce computational costs and thus energy costs. Their focus is on reducing the data in the early stages of their 8 stages pipeline and offload the heavy computations to a server. Yan *et al.* [40], like Raffa *et al.*, focus on reducing the data by adaptively changing the sampling frequency on an Android smartphone. By reducing the sampling frequency when possible, they reduce the total computational cost of recognizing activities, reducing the energy costs by 45 % to 55 %.

3 Continuous Gesture Recognition

Continuous Gesture Recognition is the process of segmenting a continuous data stream, and performing gesture recognition on the segments. The solutions for continuous gesture recognition, consist of number of different stages, thus creating a *pipeline* for the data processing. For example the pipeline used by Raffa *et al.* [28], consists of 8 stages: (1) Sensors capture, (2) low pass filtering, (3) Gesture segmentation, (4) Early Template Matching, (5) Normalization, (6) Feature Extraction, (7) HMM + Garbage model and (8) Late Template Matching, where they use a hidden Markov Model (HMM) based gesture recognizer.

3.1 Continuous Gesture Recognition Pipeline

Inspired by [28], we propose a simplified pipeline for gesture recognition with the following 5 stages: (1) data gathering, (2) filtering, (3) segmentation, (4) normalization (5) recognition. We use this simplified pipeline so that we can better describe the process without going into details of specific algorithms, but also so that we can more easily distinguish between the segmentation and the gesture recognition algorithm stages of the pipeline.

3.1.1 (1) Data Gathering

The data for continuous gesture recognition is a continuous data stream $D = \{data_t \mid t \in \mathbb{N}\}$, generated by *e.g.* an accelerometer.

3.1.2 (2) Filtering

Filtering helps reduce noise in the data, *e.g.* from hand trembling. There exists a range of different filtering algorithms. One such filtering algorithm, used by [28], is the Exponential Moving Average (EMA), which uses the average of the previous x data samples to smooth the data stream. This stage is not strictly necessary, but can help increase the accuracy, as we show in Section 6.

3.1.3 (3) Segmentation

Like filtering, the segmentation can be done using a variety of different methods. A common method is to use *thresholds*, where you have *e.g.* a high and a low threshold. The data can easily be segmented using this:

1. When a data sample at time t, $data_t$, is above the high threshold, it indicates that a gesture is here
2. We use the low threshold to find the start and end of the gesture:
    - The start is the earliest point $data_{t-n}$ before time t, below the low threshold
    - The end is the first point $data_{t+m}$ after time t, below the low threshold

The problem with this simple method is that if just one data sample is below the low threshold, the start/end is selected. The segmentation method proposed by [28] uses 3 thresholds:

1. A high threshold HF_{tS}
2. A low *forward* threshold HF_{tF}
3. A low *backward* threshold HF_{tB}

The *HF* prefix of the thresholds is short for *Hand Force*. Beside the thresholds, the algorithm also enforces a *temporal* constraint T_t (*e.g.* 200 ms). The algorithm proposed by [28] transforms the 3D accelerometer data stream to a 1D data stream by calculating the hand force as

$$HF = |(\sqrt{x^2 + y^2 + ^2} - gravity| \tag{1}$$

where x, y, and z are the accelerometer data values for each sample, and $gravity = 9.80665$ is a constant. This segmentation works much like as previously described, but with more thresholds and the temporal constraint it works as follows:

1. When a data sample at time t, $data_t$, is above the high threshold, HF_{tS}, it indicates that a gesture is here
2. We use the weak thresholds, HF_{tB} and HF_{tf}, and the temporal constraint, T_t, to find the start and end of the gesture:
    - The start is the latest point $data_{t-n}$ before time t, where this and the *earlier* data points within duration T_t are below HF_{tB}
    - The end is the first point $data_{t-n}$ after time t, where this and the *later* data points within duration T_t are below HF_{tF}

An illustration of how the thresholds are used to find the start and end can be seen by Figure 1. For more details on how the segmentation algorithm works, see [28].

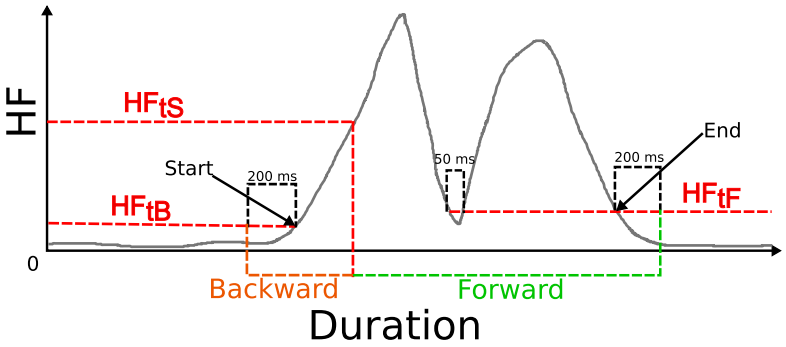

Figure 1: The segmentation algorithm with a temporal threshold of 200 ms. *Figure is inspired by Figure 2 in [28].*

3.1.4 (4) Normalization

The normalization is part of the gesture recognition. Normalization can be used to ensure that the gesture segments share the same *lengths* to properly compare them. The length of a gesture segment is the number of data samples contained in the segment and varies depending on the duration of the gesture.

Besides normalizing the lengths, the minimum and maximum values of the gesture segments can also be rescaled to the same maximum and minimum value. This stage is not strictly necessary, depending on the gesture recognition algorithm, but can help increase the accuracy, as we show in Section 6.

3.1.5 (5) Recognition

This stage performs the gesture recognition. As mentioned, there are numerous algorithms to do this stage. In the following section we will describe two such algorithms and the general idea of gesture recognition.

3.2 Gesture Recognition

This section provides insight to which techniques for gesture recognition exists and how well they perform. The results of this section, will be the base for choosing which algorithms to measure energy on and to optimize for better energy consumption.

As mentioned, a number of solutions for gesture recognition on accelerometer (and gyroscope) sensor data already exists. Niezen and Hancke [24] evaluate three different techniques for gesture recognition. They analyze gesture recognizers based on a hidden Markov model (HMM) [36], an artificial neural network (ANN) [34] and a dynamic time warping (DTW) technique [29]. Some of their findings are summarized in Table 1.

Technique	Accuracy	Execution time
HMM (4 states)	66.25 %	2.8 ms + 10.5 ms
HMM (8 states)	96.25 %	2.8 ms + 12.2 ms
ANN	90.00 %	1.8 ms + 23.02 ms
DTW	96.25 %	8.31 ms

Table 1: Accuracy and execution time results from [24]. The HMMs and the ANN have additional execution time because they require preprocessing time in addition to recognition time.

The results from Table 1 shows that DTW is a good candidate for an energy efficient gesture recognizer. Our hypothesis for this is that *lower execution times will mean lower energy consumption*, based on the reasoning that each method will utilize the CPU and memory equally for the same time period, and lower time periods should result in lower energy consumption.

However, while the DTW technique has the best performance by this evaluation, a paper by Webbrock *et al.* [38] introduces a simple gesture recognizer called the $1 Recognizer. They compare $1 with DTW and find that $1 performs roughly 80 times faster than DTW, with the same or higher accuracy. This does, however, not mean that the simple, fast and yet accurate $1 recognizer is the best for the job.

Herold and Stahovich [14] present a performance optimized version of the $1 recognizer, called the 1¢ recognizer. They evaluate the 1¢ recognizer to perform up to 80 times faster than the $1, at the cost of just 1.5 % accuracy.

This analysis has shown that while methods from machine learning (*e.g.* HMM, ANN), can provide high accuracy, they can also be expensive to run. We decide to implement and evaluate a DTW based gesture recognizer and the 1¢ gesture recognizer, and will in the following sections describe both of these algorithms. We have decided on these algorithms as they are both easy to implement, but also because DTW is a widely used technique and 1¢ is a very optimized gesture recognizer. By implementing these two gesture recognizers, we can compare them and show how 1¢ performs compared to DTW in terms of energy consumption.

3.2.1 1¢ Recognizer

The 1¢ recognizer [14] is a simple gesture recognizer, based on the $1 gesture recognizer. We have chosen to evaluate this recognizer as the authors claims that is very accurate and, more importantly, very fast, thus suited for battery-powered or low power devices. The algorithm can be described as:

1. Resample gesture segments to a fixed size N, using piecewise linear interpolation
2. Transform the 3D gesture segments to 1D segments, by calculating the distance between the centroid and each of the 3D data points
3. Z-normalize the transformed 1D data
4. Calculate the distance between the input trace and *all* of the traces in the gesture library, using the Euclidean distance, and return the gesture with the lowest distance

The time complexity of the 1¢ gesture recognizer is $O(m \times n)$, where m is the number of traces in the library, and n is the length of the longest gesture segment.

The traces in the gesture library have also, when inserted, been resampled and transformed. For the complete pseudocode, see Appendix C or [14].

The algorithm described in [14] is designed for 2D data, where we have 3D data from the 3D accelerometer. To support this, we make a few changes to the pseudocode, but fortunately the algorithm is easy to extend to 3D, which is another advantage compared to the $1 on which it is based, as the $1 is not easily modified for 3D.

3.2.2 Dynamic Time Warping (DTW)

The Dynamic Time Warping (DTW) algorithm [29] compares two time series, even if they are of different lengths. As mentioned in Section 3, DTW is much slower than the

1¢ recognizer, but it is also simpler to implement. Due to this, there have already been a lot of optimizations and approximations to the DTW algorithm [16, 30], but we have chosen to implement the original unoptimized algorithm.

For two time series $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_m\}$, the distance between X and Y can be computed based on dynamic programming using this formula:

$$DTW(X, Y) = D(n, m) \tag{2}$$

where

$$D(i, j) = d(x_i, y_j) + \min \left\{ \begin{array}{l} D(i, j-1) \\ D(i-1, j) \\ D(i-1, j-1) \end{array} \right\} \tag{3}$$

and where $d(x_i, y_j)$ is a distance function such as Euclidean distance.

We use DTW as a gesture recognizer by comparing the input gesture trace with each gesture trace in the library, exactly like the 1¢ gesture recognizer. Thus the time complexity of the DTW gesture recognizer is $O(m \times n^2)$, where m is the number of traces in the library, and n is the length of the longest gesture segment.

3.2.3 1¢ and DTW Comparison

Table 2 shows an accuracy and execution time comparison of DTW and 1¢. The execution time is the average time used to recognize a single gesture, in a gesture library with 100 traces, *i.e.* 100 comparisons are performed. The execution time and accuracy are from our implementations. The gesture set and implementation details are described in Section 6.

Technique	Accuracy	Execution time
DTW	89 %	189.4 ms
1¢	92 %	0.17 ms

Table 2: Accuracy and execution time comparison of 1¢ and DTW

4 Energy Optimization

The primary goal of software optimization is to improve the performance, typically runtime, of the program. By reducing runtime we can also reducing energy consumption as shown by [3]. Optimizing for energy, or runtime, can be done in different ways and at different abstraction levels.

4.1 Optimization Abstraction Levels

We define and describe the following three abstraction levels for software optimization:

1. Frequency and voltage scaling
2. Compiler, language and hardware specific
3. Algorithmic

Optimizations on these levels are not mutually exclusive, and optimizations on each level can and should be combined for the best results. However, due to the scope of this thesis, we will only focus on one of these.

4.1.1 Frequency and Voltage Scaling

Frequency and voltage scaling is the lowest abstraction level of optimization as this includes changing how the hardware performs.

By scaling the frequency and/or voltage of a CPU, we change the computational power of the CPU, but also the energy per instruction. The dynamic power consumption P of a CPU is approximately

$$P = ACV^2F \tag{4}$$

where A is the activity factor, *i.e.* the fraction of the circuit that is switching (typically $\frac{1}{2}$), C is the capacitance, V is the voltage and F is the clock frequency [35, p. 184]. The power, and thus the energy per instructions, grows quadratics in terms of voltage.

In theory this means that the lower power, and thus frequency, the CPU uses, the less energy per instruction is consumed. However research by De Vogeleer *et al.* [6] shows that there exists a *Energy/Frequency Convexity Rule*, which means that if you plot the energy per instruction, you would see a convex curve rather than a quadratic curve. In other words, there exists a optimal CPU frequency with regards to energy per instruction, where lower frequencies results in higher energy per instruction. See [6] for an analysis of this.

Scheduling can then be used to schedule applications to run with the optimal CPU frequency and thus reduce the energy consumption. Yuan and Nahrstedt did this and achieved up to 72 % energy reduction [41].

4.1.2 Compiler, Language and Hardware Specific

The next level of optimization is exploiting compiler and/or language features.

Examples of this are [3, 32] who obtain up to 90 % and 77 % energy reduction by rewriting code pieces. One such example mentioned by [3] is to use *inlined* functions to avoid the function call overheads, where [3] found 16.89 % energy savings by rewriting Listing 1 to Listing 2.

```
int square(int x){
        return x * x;
}
```

Listing 1

```
#define square(x) (x*x)
```

Listing 2

While effective, these types of optimizations are based on *how* algorithms are implemented, and may depend heavily on certain programming languages or compilers, as well as the code being optimized. For more examples of how *e.g.* C code can be optimized, see [15].

It is also possible to optimize the runtime of a program by knowing and exploiting *e.g.* the cache size. One such example is *loop blocking* (which is just one of many loop optimizations), where additional loops are created to improve locality and cache reuse. Listing 3 shows an example of this.

Another specific hardware optimization is using the Streaming SIMD extensions (SSE) instruction set, which uses the XMM registers. By using SSE, you can perform the same instruction, *e.g.* addition, to multiple data, as fast as a single instruction to a single data. As the XMM registers are 128 bit, you can thus increase perform four-fold if your data is 32 bit.

```
//Original code:
for i = 1 to n do
  for j = 1 to n do
    a[i; j] = b[j; i];
  end for
end for
```

```
// Loop blocked code:
for ii = 1 to n by B do
  for jj = 1 to n by B do
    for i = ii to min(ii + B - 1; n) do
      for j = jj to min(jj + B - 1; n) do
        a[i; j] = b[j; i];
      end for
    end for
  end for
end for
```

Listing 3: Loop blocking where B^2 elements fits in L1 cache. Example from [18]

4.1.3 Algorithmic

The highest abstraction level of optimizations happens at the algorithmic level. Often when algorithms are optimized, they are optimized with regards to their complexity, denoted by *e.g.* the big-O notation.

However when we are optimizing for energy consumption, we need to consider the actual run time as two algorithms with the same big-O complexity may have very different energy consumptions. One example of an algorithmic optimization, where the big-O complexity stays the same, is the optimization [14] (1¢) did the the $1 gesture recognizer from [38], where they, among other things, removed the rotation (an expensive operation), and reduced the runtime by a factor 80.

The advantage of optimizing at this level compared to the others, is that the optimizations are general and not dependent on certain hardware, compilers or languages. We will thus focus on optimizing the algorithms of continuous gesture recognizers.

Some of the optimization we perform can considered begin at either algorithmic or compiler/language level. We do, for example, want to pay some attention to certain operations that are commonly expensive to perform across multiple programming language, compiler or even processor architecture (*e.g.* ARM, Intel x86, *etc.*), such as computing square root.

4.2 Optimization Techniques

We utilize the following 5 optimization techniques:

1. Avoid expensive operations
2. Perform mathematical reductions
3. Utilize approximations
4. Reduce the amount of data
5. Change data types

4.2.1 Expensive Operations

Certain operations are computationally expensive to perform on the CPU, almost regardless of the architecture, and are thus more time consuming. Based on a series of tests, Agnor Fog has compiled a table (see [9]) of the performance of different instructions several Intel, AMD and VIA CPUs.

Two very common operations found in the gesture recognition algorithms are division (`IDIV`) and square root (`FSQRT`). As Table 3 shows, these operations are much more expensive than *e.g.* `ADD` and `IMUL`. The operations should not necessarily be avoided, but it should be kept in mind that they are expensive to perform, keeping the use of them to a minimum could improve performance.

Instruction	Latency	Reciprocal throughput
`ADD/SUB`	1	0.25
`FADD/FSUB`	3	1
`IMUL`	3	1
`IDIV`	22-29	6
`FSQRT`	10-23	4-9

Table 3: Example instructions and their latency and throughput on a Intel Haswell CPU using 32-bit registers (where relevant). The unit in both columns is core clock cycles. See [9] for full table.

4.2.2 Mathematical Reduction

Mathematical reduction at a algorithm level is another way to reduce the computational cost of an algorithm. An example is the Exponential Moving Average (EMA) filter used by [28]:

$$S_t = \alpha \times X_t + (1 - \alpha) \times S_{t-1} \tag{5}$$

Where S_t is the value of the EMA at time t, α is a constant coefficient between 0 and 1, and X_t is the value to filter.

To calculate S_t we thus have to perform 2 multiplications, 1 addition and 1 subtraction. We can rewrite Equation (5) as Equation (6):

$$S_t = S_{t-1} + \alpha * (X_t - S_{t-1}) \tag{6}$$

By doing so, we eliminate one multiplication of the equation, which, as shown by Table 3, can have a decent effect on the computational cost. Unfortunately [9] does not include measurement for floating point multiplication (`FIMUL`), so we cannot calculate the actual savings in core clock cycles, but based on Table 3 and the integer multiplication, we can remove 3 out of 8 cycles to a total of 5 cycles, effectively reducing the latency of Equation (5) by 37.5 %. Since Equation (5) is performed for each data sample, the savings are significant.

4.2.3 Approximations

A lot of performance increase can come from approximations, but typically with a side effect, such as loss of accuracy. Approximations should only be used if the loss of accuracy is acceptable, and where the execution time is excessive. One example of an approximation that we tested and showed a 12.96 % performance increase was approximating the instantaneous Hand Force (HF) of [28]. We can write an approximation to

$$HF = |\sqrt{x^2 + y^2 + ^2} - gravity| \tag{7}$$

as

$$HF = |x^2 + y^2 + z^2 - gravity^2| \tag{8}$$

While Equation (8) does not give the same result as Equation (7), the result works as an approximation for segmentation as can be seen in Figure 2. The thresholds for the optimized equation (Equation (8)) need to be modified to capture the same information, but as we will show in Section 6, it is possible to use this approximation.

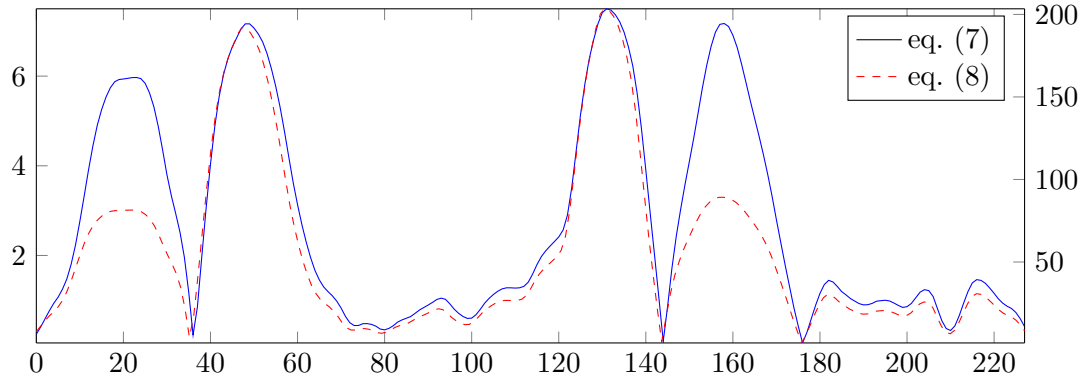

Figure 2: Hand force for drawing a square with Equations (7) *and* (8).

4.2.4 Data Reduction

The size of the data have a direct effect on the performance, as each data point is used as input to several of the functions in the gesture recognition algorithms. Effectively, by reducing the number of data points by *e.g.* 50 %, we reduce the computational cost by 50 %. To reduce the size of the data we typically have to perform some additional computations. One such example is Equation (7) where we turn each 3D accelerometer data sample into a 1D data point, thus reducing the size of the data to one-third of the original size. Data reduction also reduces the memory required, assuming that the original data is not stored.

4.2.5 Data Types

Data types are found in most programming languages. Examples of data types are `float`, `integer`, `char`, `short` and `double`. The reason why we mention data types, is that the data types can have an effect on the performance of the program (both memory and runtime). A common way to optimize the code regarding to memory consumption, is to use the minimum needed size for the task, *e.g.* short (2 bytes) compared to integer (4) in the C programming language. However, as pointed out by [15], data types that does not fit in a *word* (typical 32 or 64 bits on never processors), may have a negative impact on the runtime, as the CPU may need to convert these to and from the word size.

Furthermore, as shown by Table 3, operations involving floating point number types, *e.g.* `float` and `double`, are often more expensive to perform that they fixed number (integer) counterparts.

5 Energy Measurement

In Section 2 we mentioned a number of methods and tools for measuring energy of a program.

We have tried and tested some of the internal energy measurement tools mentioned:

1. EACOF: Difficult to understand how to use it and required 3 executables to run. Rejected due to complexity and lack of documentation on how to use it.
2. JouleUnit: Only possible to use through Java with a workbench for the Eclipse IDE, and contains little to no documentation. Rejected due to Java constraint.
3. PowerTutor, Intel Performance Monitor, Dr Power (Android applications): Outdated applications for Android with limited accuracy; not possible to log the measurements to a file; on. Rejected due to not being developer friendly.
4. RAPL: Easy to use and developer friendly, but limited to newer Intel CPUs.

We decided not to try any of the external hardware tools, as these require extra time and money investments.

We have decided to use the Running Average Power Limit (RAPL) interface to measure energy consumption.

5.1 Running Average Power Limit (RAPL)

The RAPL interface was created to work with the Intel Turbo Boost technology, where it would provide energy and power information about the CPU, DRAM and other uncore devices. The RAPL interface consists the following domains:

- Package, which in turn consists of the following two domains:
  - PP0 (Core Devices)
  - PP1 (Uncore Devices)
- DRAM (only available on Haswell and newer processors)

Core devices are the processors components involved in executing instructions, such as the processor core and the caches. Uncore devices are devices close to the CPU but is not part of the chipset, such as *e.g.* GPUs. We will use measurements from the PP0 (CPU) and DRAM domains.

The accuracy of RAPL have been analyzed in [23, 12, 7], where they show RAPL to be very accurate, but unfortunately they do not provide any numbers. The analysis in [12] mentioned an inaccuracy in the form of an offset which they attributed to RAPL not measuring DRAM, but when this was accounted for they reach a average error rate of only 1.12 %. However, as mentioned, newer chips (Haswell and newer) are also able to measure the energy of DRAM. See Appendix E for a list of support Intel CPU architectures.

The disadvantage of using RAPL is that it is *only* available on newer Intel CPUs, where most wearables, *i.e.* devices intended for the our energy optimization, typically have an ARM processor, which do not, to our knowledge, contain any interface such as RAPL. Thus the energy measurement method chosen in this thesis, cannot provide actual measurements on ARM-based wearables, but may still be useful for measuring optimizations of algorithms.

On Linux the RAPL measurements can be read in the following three ways:

- Reading files in the `/sys/class/powercap/intel-rapl` directory
- Using the `perf` or `papi` tools (requires root/sudo)
- Reading from the Model Specific Registers (MSRs) (requires root/sudo)

The disadvantage of using `perf` or `papi` is that they will read the energy during the entire runtime of the process, and is thus not possible to read energy between specific code pieces. Furthermore, the RAPL values are only updates once every 1 ms. We found the easiest way to read RAPL on Linux was to read the files in `/sys/class/powercap/intel-rapl`. To read the energy consumed by a piece of code, we read this value at the beginning and the end of the code we want to measure, and then compare the two values. The unit read from this file is microjoule (µJ).

For Mac OS the only way we found to read RAPL was to read it directly from the MSRs or by using `perf`. Windows does not (yet) support RAPL.

6 Evaluation

We evaluate and optimize different algorithms in this section, using the optimization techniques presented in Section 4 and the energy measurement method presented in Section 5.

6.1 Evaluated Algorithms

We have chosen to implement, evaluate and optimize the following algorithms:

1. The segmentation algorithm from [28]. We do not implement the HMM gesture recognizer from that paper, because from Section 3 we see that HMMs perform worse than DTW (and thus 1¢) and typically are more time consuming to implement. We chose to implement this particular segmentation algorithm because it is claimed to be optimized and be energy efficient.
2. The 1¢ recognizer from [14]. We chose to implement this particular gesture recognition algorithm because it is highly optimized, and we thought it would be interested to see if we could optimize it even further.
3. A gesture recognizer using the DTW algorithm. This algorithm is, unlike the other two, not optimized. We chose to implement DTW because it is widely used and easy to implement, but also so that we could examine which architectural features that differs from 1¢ and DTW.

6.2 Implementation

We implemented the three algorithms ourselves in C and compiled with `gcc` 5.3.0 with no compiler optimization (`-O0`). We did this to ensure that the algorithms was implemented as efficiently as possible prior to our optimizations, as well as sharing the same quality of code. Noteworthy implementation details:

- Avoided using expensive standard library function such as `pow(a,2)` where possible. Instead we calculate *e.g.* `pow(a,2)` as `a * a` (more than twice as fast).
- Only used 32 bit data types where possible, *i.e.* `float` and `int`.
- DTW has been implemented as iterative rather than recursive to improve performance

6.3 Gesture Set

We have tested the optimizations on the gestures shown in Figure 3. Since some of them are hard to illustrate on paper, short descriptions the gestures can be found in Table 4.

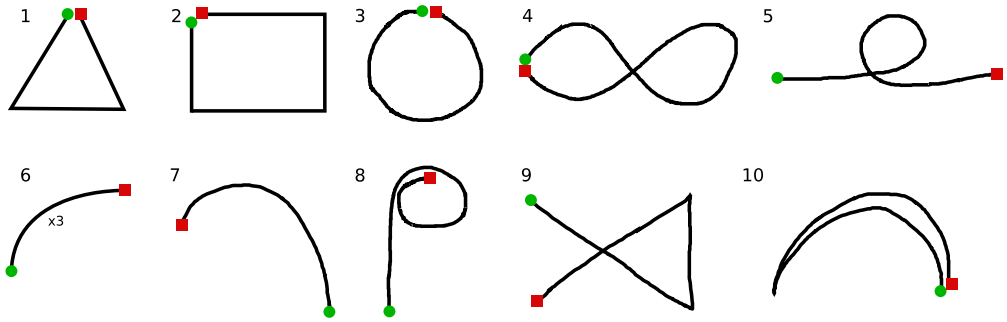

Figure 3: Gesture set. Green dot indicates starting point, where the red square indicates the end.

ID	Name	Description
1	Triangle	A triangle
2	Square	A square
3	Circle	A circle
4	Infinity	An infinity (∞) symbol
5	Loop	A loop from left to right
6	Knock	Knock three times
7	Throw Back	Move hand from right hip over left shoulder in a throwing motion
8	Lasso	Move hand from right hip up over right shoulder and perform a lasso motion
9	X	A cross (X)
10	Roll	Slowly rotate the arm 180 degrees, and then back again

Table 4: Gesture Descriptions for gestures in Figure 3

This gesture set draws inspiration from some of the gestures found in the gesture recognition algorithms analyzed, and is based on being fast and *easy* to do in 3D. The gestures from *e.g.* the 1¢ paper are hard to do in 3D, but easy to do on paper/tablet in 2D.

Furthermore, some of the 3D gestures found in other papers, are inversed or rotated duplicates of other gestures in the same gesture set. These gestures will not cause a problem for *e.g.* DTW, but since 1¢ gesture library is based on the distances to the centroid of the gesture, the same gesture reversed or rotated will result in the same centroid, and thus the same 1D gesture trace in the library, which makes it impossible to use these as *unique* gestures.

6.4 Setup

We perform the tests on a laptop with a Intel i7-5500U CPU, with 2 (4 with hyper-threading) cores running at 2.4 GHz, a 128 kB level 1 cache, a 512 kB level 2 cache and a 4096 kB level 3 cache. This CPU is based on the Broadwell architecture, and allows us to fully use the RAPL interface with DRAM measurements as described by Section 5. Furthermore the laptop was equipped with 16 GB of RAM and a 256 GB Samsung MZ7LN256 SSD. The laptop is running 64 bit Arch Linux with Linux kernel 4.5.1.

The device we use for testing outperforms typical wearable or embedded devices substantially, especially the cache size differs a lot from what is typically found in these devices. However, the number of instructions executed should be similar to running the programs on a less powerful device, even if it has a CPU with an ARM architecture.

To reduce noise in the measurements, we perform the following steps on the laptop:

1. Disable Intel Turbo Boost and heat management (both throttles the CPU) in the BIOS settings, such that we have a constant CPU frequency (2.4 GHz)
2. Kill and stop any unnecessary process and service, including desktop environment and network
3. Empty the cache between each iteration

Listing 4 shows pseudocode describing how we use RAPL to measure the energy consumption of code pieces.

```
main(){
 <some code>
 start = read_energy();
 <code to measure>
 end = read_energy();
 energy_used = end - start;
 <some code>
}

read_energy(){
 cpu_energy = read("/sys/class/powercap/intel-rapl:0:0/energy_uj");
 dram_energy = read("/sys/class/powercap/intel-rapl:0:2/energy_uj");
 return cpu_energy + dram_energy;
}
```

Listing 4: Pseudocode for measuring energy using RAPL. The numbers in the file path indicates which CPU (in this case CPU 0) and domain to read. RAPL domains are: Core devices (CPU, caches, etc.) = 0, uncore devices = 1, DRAM = 2.

The data used for the measurements are generated using an Android application, which collects accelerometer data from a Motorola Moto X (2nd generation), running Android OS 6.0 (API level 23). The accelerometer data is collected as floating point numbers (as defined by Java) by the accelerometer in the Android `SensorManager`.

The training training, *i.e.* the data for the gesture library, consists of 10 training traces for each of the 10 gestures in the gesture set from Section 6.3, giving a total of 100 traces in the library. The average number of samples per gesture is 207, the maximum is 302 (a "Square" gesture) and the minimum is 159 (a "Roll" gesture). The data is saved on the laptop's SSD, so that the laptop can read and process it.

The input data used for the test is generated in a similar manner. We generate 10, 1 min long input traces with 10 randomly chosen gestures performed in that time. These 10 input traces are then aggregated in a single 10 min input trace, and saved on the laptop's drive. However since the laptop used for these tests can process this 10 min trace in a few milliseconds, and the RAPL energy measurements are only updated once every 1 ms, we aggregate the 10 min trace 10 times to create a 100 min trace, and thus increase the processing time to get more accuracy results. For future reference, we will call the 10 min trace the *short* trace and call the 100 min trace the *long* trace.

The input data is collected by only one person and is done while this person was standing still and performing gestures roughly once every 6 s (10 gestures in 1 min).

For all the energy consumption measurements we subtract the background energy of the CPU, *i.e.* the energy consumption of the CPU in an idle state. This is done

by measuring the idle energy consumption over 1 min, and then subtract the appropriate amount depending on how must time was spend by the segmentation and gesture recognitions.

6.5 Segmentation Optimization Methods and Results

We have chosen to use the segmentation algorithm from [28]. The segmentation algorithm perform two expensive operations: applying a filter to the data and computing the hand force. We set the coefficient of the EMA filtering algorithm $\alpha = 0.2$ and the temporal threshold to be 200 ms. The weak and strong thresholds will vary depending on the optimization being done. We perform three unique optimizations to the segmentation algorithm: Optimize the filter, change the hand force, and remove the filter entirely.

The algorithm applies an Exponential Moving Average (EMA) filter to the data to reduce the noise from subtle hand movements. The filter equation reported in [28] is:

$$S_t = \alpha \times X_t + (1 - \alpha) \times S_{t-1} \tag{9}$$

Where S_t is the value of the EMA at time t, α is a constant coefficient between 0 and 1, and X_t is the value to filter. We perform a mathematical reduction optimization (see Section 4) to this equation and get this equivalent, but cheaper, equation:

$$S_t = S_{t-1} + \alpha \times (X_t - S_{t-1}) \tag{10}$$

The segmentation is performed on a 1D data steam, rather than the 3D from the accelerometer, by computing the *hand force* of the accelerometer data. The hand force equation from [28] is

$$HF = |\sqrt{x^2 + y^2 + ^2} - gravity| \tag{11}$$

where *gravity* is a constant = 9.80665. We optimize this calculation by computing an approximation:

$$HF = |x^2 + y^2 + z^2 - gravity^2| \tag{12}$$

The removal of square root decreases the runtime of this function, and thus the energy used. To compensate for this change, we also change the the *gravity* to be squared. However, since this is a constant, there is no additional computations to this change.

The last optimization, *i.e.* the removal of the filtering, is a more drastic change. However, in our implementation, the gesture recognizers use the raw 3D data, and the filtered data is thus only used for the hand force function, and thus the segmentation. We could filter the 3D data used by the 1¢ recognizer, but this will add, and not reduce, the runtime of the algorithm. This input trace for this is the aforementioned long trace.

As Table 5 shows, we can achieve large savings by changing the hand force and removing the filter. Changing the filter is a easy optimization with no side-effects, whereas optimization (2) and (3) requires finding new correct thresholds. Notice that the accuracy changes depending on the optimization technique. The reason for this is the thresholds used by the segmentation algorithm, which needs to be fine tuned. It is our belief that the loss, or increase, in accuracy from these optimizations, can be changed if more time is put into choosing the right thresholds for this segmentation algorithm, as the segments sent to the gesture recognizers are affected by this.

ID	Method	Accuracy	Energy	Saved
0	Baseline	92	201.78 mJ	N/A
1	Change filter	92	193.51 mJ	4.10 %
2	Change hand force	93	175.63 mJ	12.96 %
3	Remove filter	83	103.69 mJ	48.61 %
4	1 and 2	93	173.60 mJ	13.97 %
5	1 and 3	85	96.53 mJ	52.16 %

Table 5: Segmentation algorithm optimizations on the long trace. The accuracy is from using the 1¢ recognizer.

6.6 1¢ Optimization Method and Results

We set the number of data points of the resampled array $N = 124$, as we empirically found this to give the highest accuracy, while still being as low as possible. The algorithms performance depends on this number, but as it is very data dependent, we will *not* try to find the optimal N.

The most expensive operations in this algorithm are the distance functions which used the Euclidean distance (also known as the L2 distance). The algorithm uses two distance functions: One for calculating the distance between two 3D points, and one for the distance between two 1D points.

6.6.1 Distance Function Optimization

To optimize and thus reduce the energy used by the distance functions, we implemented and tested a few alternatives. As mentioned, there are *two* functions that calculate the Euclidean distance: One for calculating the distance between two 3D points, and one for the distance between two 1D points. Because the functions have different input and different purpose, we cannot apply all of the optimization techniques to each function. We have tested using the following distance functions:

1. Euclidean distance (also known as L2, used as baseline)
2. Squared Euclidean distance (omit computing the square root)
3. Octagonal Boundary (distance approximation for 2D, modified to work with 3D) [2]
4. Taxicab distance (also known as L1)

See Appendix B for the definitions of these distance functions.

The results of these optimization can be seen in Table 6. The input trace for 1¢ is the long trace. As the table shows, it is possible to save up to 4.73 % energy by doing a simple approximation. However, most of the optimizations shown here also lowers the accuracy.

6.6.2 No Normalization

One way to avoid expensive operations is to remove them where possible, as shown by Table 5 where removing the filter, reduced the energy cost with loss of accuracy. In the same manner, we try to omit the normalization that is performed before the recognition. By doing so we get a energy consumption of 1148.39 mJ (1.05 % saved), but an accuracy reduction to 89 %. The ID of this method is 14.

ID	Method	Accuracy	Energy	Saved (%)
0	Baseline	92 %	1160.52 mJ	N/A
6	3D Octagonal	90 %	1161.53 mJ	−0.09 %
7	3D Taxicab	82 %	1124.51 mJ	3.10 %
8	1D Squared Euclidean	92 %	1150.12 mJ	0.90 %
9	1D Taxicab	88 %	1105.64 mJ	4.73 %
10	6 and 8	90 %	1157.48 mJ	0.26 %
11	6 and 9	89 %	1110.79 mJ	4.29 %
12	7 and 8	82 %	1116.48 mJ	3.79 %
13	7 and 9	76 %	1103.00 mJ	4.96 %

Table 6: Optimization results for distance functions with the long trace. The first row is the baseline the results are compared to.

6.6.3 Data Reduction

The previous section used approximations to reduce energy use. This section describes the result of performing data reduction optimization.

In 1¢ when we try to recognize a gesture, we compare it with every gesture trace in the gesture library. If we reduce the number of gestures in the library, we can very effectively reduce the energy consumption. In our library we have 10 training traces for each of the 10 unique gestures, giving a total of 100 traces in the library. Instead of using these 10 training traces for each unique gesture, we compute a new *average* gesture trace for each of the unique gestures. This can easily be done since all the test traces have the same length (due to the resampling). Each data point in the average gesture trace is computed as the average data point across the 10 training traces.

By reducing the size of the library tenfold the energy consumed is 273.04 mJ which is a 76.47 % saving. However, this reduction also decreases the accuracy to 74 %. This ID of this method is 15.

6.6.4 Early Termination

Rather than removing data from the library, we can also reduce the number of comparisons by terminating the comparisons when the distance reaches below a certain threshold. When the distance between two gestures is lower than this threshold, we consider it correct and do not compare the input gesture with the rest of the library. If no gesture is below the threshold distance, it will still compare the input gesture with the entire library.

This optimization results in a total energy consumption of 890.84 mJ which is a 23.24 % saving with no accuracy lost. By tweaking the threshold the energy reduction can be improved, at the cost of accuracy. We decided that the best result would be to set the threshold such that we do not lose any accuracy. The ID of this method is 16.

6.7 Dynamic Time Warping Optimization Methods and Results

To optimize DTW, we can use some of the same optimizations used for 1¢, but also use the Squared Euclidean distance optimization for 3D segments here. Since DTW runs significantly slower than 1¢, we use the *short* trace for DTW.

ID	Method	Accuracy	Energy	
0	Baseline	89 %	133.29 J	N/A
6	3D Octagonal	89 %	116.57 J	12.54 %
7	3D Taxicab:	87 %	110.31 J	17.24 %
15	Reduce library size	78 %	10.10 J	92.42 %
16	Early termination	89 %	128.62 J	3.51 %
17	3D Squared Euclidean	91 %	127.38 J	4.44 %

Table 7: Optimization results for DTW for the short trace. The first row is the baseline the results are compared to. Note that, unlike 1¢, the unit here is Joule instead of millijoule.

It is interesting that changing the distance function to squared Euclidean (ID 17) not only saves energy, but also increases the accuracy.

6.8 Optimization Summary

Table 8 shows a summary of the optimizations performed.

Segmentation		
ID	Accuracy	Saved
0	92 %	N/A
1	92 %	4.10 %
2	93 %	12.96 %
3	83 %	48.61 %
4	93 %	13.97 %
5	85 %	52.16 %

1¢ Gesture Recognition		
ID	Accuracy	Saved
0	92 %	N/A
6	90 %	−0.09 %
7	82 %	3.10 %
8	92 %	0.90 %
9	88 %	4.73 %
10	90 %	0.26 %
11	89 %	4.29 %
12	82 %	3.79 %
13	76 %	4.96 %
14	89 %	1.05 %
15	74 %	76.47 %
16	92 %	23.24 %

DTW Gesture Recognition		
ID	Accuracy	Saved
0	89 %	N/A
6	89 %	12.54 %
7	87 %	17.24 %
15	78 %	92.42 %
16	89 %	3.51 %
17	91 %	4.44 %

Table 8: Summary of optimization results for the three algorithms.

The results from optimizing the segmentation algorithm show significant reduction in energy consumption. We believe that the loss of accuracy is caused by sub-optimal thresholds, which we believe can be tuned to improve the accuracy.

Surprisingly one optimization (6) performs slightly worse than the baseline for 1¢. One reason for this is that octagonal distance function's complexity grows exponentially with regards to dimensions. The original octagonal distance function is fast approximation to 2D Euclidean distance, but our modified 3D implementation requires more computation and is thus more expensive to perform. However since DTW shows a 12.54 % saving, we believe this is likely due to variance in the measurements.

The most promising optimization is the reduction in the size of the library and the early termination. While our (naive) approach to the smaller library gives a significant worse accuracy, we believe it is possible to get near the same energy reduction with less loss of accuracy. The early termination can be modified for more energy reduction, with a loss of accuracy, making it a flexible optimization.

We have omitted various other optimizations due to too high loss of accuracy with insignificant energy reductions, including but limited to: converting all numbers to integers, changing the resample function, and using only one/two dimensions in stead of three.

6.9 Effect on Battery Life

In this section we analyze how the optimizations found will affect the battery life of wearables.

The Samsung Gear S2 (smartwatch) is one of the newer and more powerful wearables. It has a 250 mA h lithium-ion polymer battery. To compare this to our measurements, we need to convert this to Joules. The lithium-ion polymer battery have a nominal cell voltage of either 3.3 V or 3.7 V. We have not been able to find the exact battery specifications of this smartwatch, so we will assume 3.3 V. We can convert mA h to Joule with the following formulas:

$$\text{W h} = \text{mA h} \times \text{V}/1000 \tag{13}$$

$$\mathit{Joule} = 3600 \times \text{W h} \tag{14}$$

$$\Rightarrow \mathit{Joule} = 3.6 \times \text{mA h} \times \text{V} \tag{15}$$

For the Gear S2 we get a battery capacity of 2970 J by using Equation (15).

The total energy consumption of the unoptimized segmentation algorithm, for 24 h is 2905.63 mJ. The energy consumption of recognizing a single gesture with the unoptimized 1¢ is 1.16 mJ and 1332.90 mJ for the unoptimized DTW recognizer. If we assume an extensive use of the gesture recognition with 100 recognitions per 24 h we get a total energy consumption of 3.02 J with 1¢ and 136.20 J with DTW per day. It is important to note that our measurements are from running the algorithms on a powerful laptop, and thus the energy consumptions may be higher than it would be on a wearable such as the Gear S2. For the Gear S2, the continuous gesture recognizer will thus consume 0.10 % to 4.59 % of the total battery every day, depending on the gesture recognizer.

For a powerful wearable such as the Gear S2, the energy consumption of such a continuous gesture recognizer is not a big issue. However, if we compare it to a less powerful wearable, such as the Jawbone UP3 fitness tracker, we get a different result. The UP3 has a battery capacity of 451.44 J (38 mA h with 3.3 V). If the wearable was running the continuous gesture recognizer, it would consume 0.67 % to 30.17 % of the total energy capacity per day. If the only energy consumption of the UP3 was the continuous gesture recognizer, it would be able to run for

- 149 days with the 1¢ gesture recognizer
- 3 days with the DTW gesture recognizer

Using the 1¢ recognizer, even when unoptimized, instead of the DTW recognizer would result in 4503 % longer battery life.

6.9.1 Battery Measurement

To verify the results from the measurements performed with RAPL on a laptop, we have implemented the algorithms on a Android smartphone, and measure the *actual*

battery cost of running these algorithms. The smartphone we used is a Samsung Nexus S (released in 2010) with Android OS 2.3 (Gingerbread), a 1 GHz single core Cortex-A8 CPU (ARMv7-A architecture), 512 MB DRAM and a 1500 mA h lithium-ion polymer battery. We have chosen to implement it on this smartphone as it is comparable in terms of performance with a wearable like the Samsung Gear S2 which has a 1 GHz dual core Cortex-A7 CPU and 512 MB DRAM. The smartphone has prior to installing the continuous gesture recognition application, been factory reset and is running in airplane mode.

We have implemented the algorithms as a background service (`IntentService`) with a `PARTIAL_WAKE_LOCK` which keeps the CPU alive while the screen is turned off. We read a sample data from the *long* trace (see Section 6.4) every 10 ms, *i.e.* the same same frequency as the data was recorded with. The running time is thus the same length as the trace, *i.e.* 100 min for the long trace. We measure the battery level before and after reading and doing continuous gesture recognition on the entire trace using the Android `PowerManager`. Unfortunately there are two issues with this:

1. The battery level is a integer between 0 and 100 %. Is it not possible to get better precision (*i.e.* 45.75 % rather than 45 %).
2. The battery consumption also includes the screen and other background services which consume a noticeable amount of energy, thus reducing the accuracy of the measurements.

To compensate for this we run this test with the long trace, such that the battery usage is maximized. The accuracy of our measurements increases as the battery usage increases, as we can only measure the battery level in integers. We also subtract the energy used for reading the file by running without any segmentation or gesture recognition. By reading the energy used by the application in the built-in battery monitor, we can also estimate how many percentages of our measured battery use was actually used by our application, and not the Android OS.

	Predicted	Measured
1¢	0.67 %	0.70 %
DTW	30.17 %	39.13 %
Factor	45.03	55.9

Table 9: Comparison of the predicted and measured battery usage. The predicted battery usages are based on RAPL measurements with 24 h *of segmentation and* 100 *gesture recognitions on the Jawbone UP3 wearable. The measured battery usages are based on measuring the actual battery level on a Samsung Nexus S with the long trace (*1000 *gestures in* 100 min*).*

Table 9 shows the predicted and the measured battery usages on the Jawbone UP3 wearable and the Samsung Nexus S Android smartphone. The methods for measurement are different, as the predicted results (from RAPL) is based on 24 h of segmentation with 100 gesture recognitions, and the measured results are from 100 min of segmentation with 1000 gesture recognitions. The reason is that since the accuracy of measuring using the battery level is low, we were not able to measure any battery usage from the segmentation algorithm, and thus we were not able to find the battery usage of 24 h of segmentation on the smartphone. However, since the primary source of energy consumption are the gesture recognizers, we believe these results still prove that the energy consumption factor of the two algorithms are very alike.

6.10 Comparative Analysis of Architectural Features

So far we have seen that DTW is much more expensive to run. In this section we analyze why. To do this, we use the `perf` Linux utility which is capable of measuring various performance counters such as cache misses, instructions, cycles, *etc.* We run and measure the performance of 100 aggregated iterations DTW and 1¢ with the short trace.

Figure 4 shows how the two gesture recognizers compare, sorted by difference in percentage, where `task-clock` is the time used in ms. Appendix D contains the data plotted in Figure 4. It should be noted that the use of `perf` adds a noticeable overhead in terms of performance, and that most, if not all, numbers reported is in reality lower than shown here.

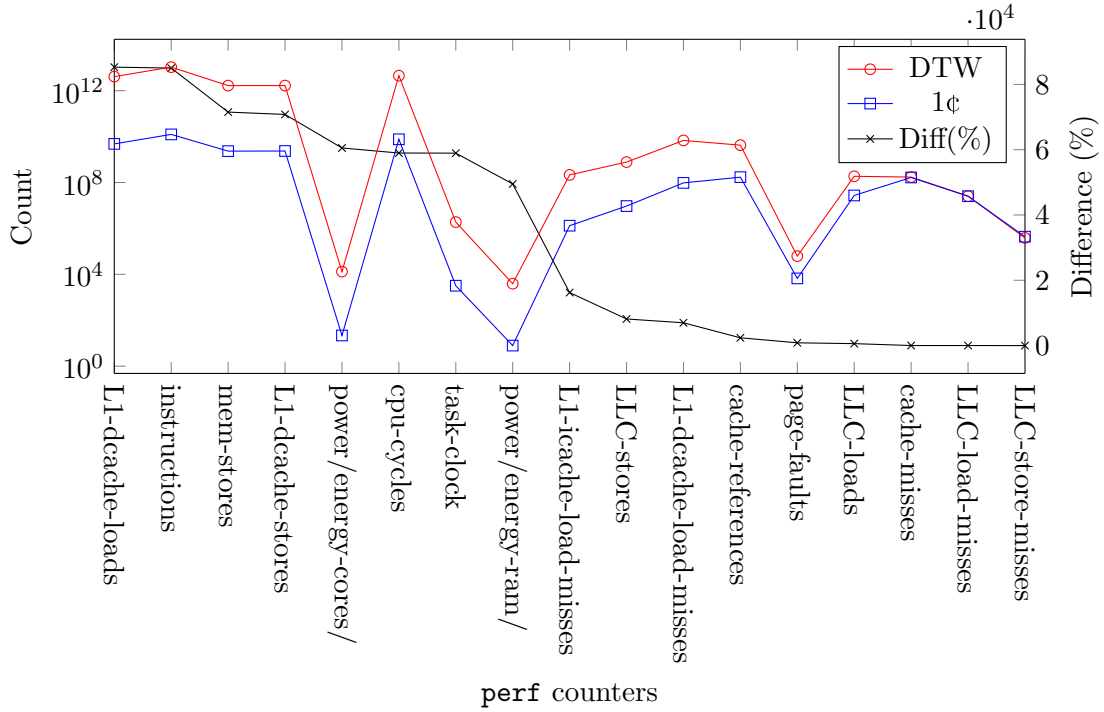

Figure 4: Graph (left logarithmic axis) showing the difference (right axis) between running DTW and 1¢ with 100 *aggregated iterations on the short trace. The counters are sorted by the difference of the counts.*

From Figure 4 we can see that the biggest difference is in L1 data cache loads, but also L1 data cache stores is one of the largest differences. This is to be expected as we also see a lot more memory stores with DTW than 1¢, since DTW calculates an $n \times m$ matrix for each gesture comparison, where n is the size of the input gesture and m is the size of the library gesture currently being compared to. Beside the memory required for this, the number of instructions in DTW also shows that this is computational expensive.

The features that contribute most to the difference between DTW and 1¢ is thus a mix of instructions, memory and cache use.

7 Conclusion

This section concludes this thesis. We answer our research questions and summarize our results, and then discuss the results, the method and future work.

7.1 Results

We showed that it is possible to accurately measure the energy consumption using the Intel Running Average Power Limit (RAPL) interface. To demonstrate how RAPL can be used, we implemented and evaluated a segmentation algorithm and two gesture recognition algorithms, and performed optimizations on an algorithmic level, not utilizing any CPU or programming language optimization techniques. The RAPL interface is just one of several other methods for measuring energy consumption, but due to its accuracy and simplicity, we conclude that this is the best method among the ones we tested. We showed that it is possible to measure differences on pieces of code, making it possible to easier measure the result of particular optimizations. By using RAPL we showed that the 1¢ gesture recognizer significantly outperforms our DTW based gesture recognizer in terms of energy consumption. We saw that a combination of instructions, CPU cycles, memory stores, data cache stores and loads were the primary architectural features that contributed most to the energy consumption of the evaluated algorithms.

We conclude that RAPL is a powerful tool for measuring energy consumption of software – even for small code pieces. Our evaluations and analyses show that measurements performed on a laptop with an Intel Broadwell architecture are comparable with measurements done on a Android Smartphone with a ARMv7-A architecture.

7.2 Discussion and Validity of Results

In our tests with RAPL we used a powerful laptop and found that the results are close to the measurements on the Android smartphone. While the results showed a noticeable difference (factor 10), this may be due to inaccuracies of the method used to measure battery usage on the smartphone – A method we have not been able to verify the correctness of. This may also be because of the significantly different cache sizes of the two devices used for measurement, as we saw that the difference between cache loads and stores are one of the major differences between the two tested algorithms. Furthermore, we were also not able to measure the energy consumption of the segmentation algorithm on the smartphone (due to limited accuracy of reading the battery level).

The way we tested the algorithms was by saving an input trace on the SSD of the laptop, read the file into memory, and then simulated the 10 min to 100 min trace as fast as possible. This is clearly not how the continuous gesture recognizers would work in a real situation. The implementation on the smartphone, however, is implemented as a real background service which handles new data samples every 10 ms. The different ways the two implementations work may also have affected the results.

We have not focused a lot on the accuracy of the algorithms, and thus not focused much on validating it. The traces recorded were all recorded by the single person, and some of the optimizing may have very different affects on accuracy. Furthermore only 100 unique gestures were recognized. The input traces were recorded while the person was standing still. If the person had been moving around, or even doing some activities such as walking while doing the gestures, the accuracy may have been reduced, and the optimizations for the segmentation algorithms may have had different results.

7.3 Discussion of Method

The tools and frameworks we have tried, aside from RAPL, are difficult to use, inaccurate or limited to a certain language/environment. One significant problem with the RAPL interface is the lack of supported platforms. Currently it is only available on Sandy Bridge (or Haswell for DRAM support) Intel CPUs, which means that the CPUs must be from Intel and newer than 2011 (or 2013 for DRAM support). This is especially a problem as most wearable or embedded devices use CPUs based on an ARM architecture, where the interface is not available. Furthermore, Microsoft Windows does not currently support the interface either, which limits the RAPL interface to only a fraction of devices being used today.

Lastly, even though RAPL is easy to use, it is not easy to isolate the energy consumption of small pieces of code, as the measurements are CPU-wide. We used several methods to isolate it, *e.g.* killing all unneeded processes, measuring and subtracting background energy and running many iterations to get the average. From our measurements we have seen a lot variance in the results even with our efforts to decrease it (see Appendix A). It is easy to use RAPL, but it is hard to get accurate and isolated results.

7.4 Future Work

One of the biggest issues we experienced with measure energy using RAPL was isolating the energy. It is not only hard to do, but it is also an inconvenience for developers. However, if we could analyze and find a correlation between energy and *e.g.* instructions, then it would be possible to isolate the energy consumption of specific processes, without killing all other running processes. By using `perf record` it is possible to measure and show events, *e.g.* instructions, by process. By doing so, it may be possible to use event/performance counters from each process to estimate the energy consumption of that process. This method is much like the instruction level analysis described in Section 2, but where RAPL is used rather than external hardware tools.

It could be imagined that an interface like RAPL would be implemented in ARM architectures, such that it is possible to measure energy consumption of devices with ARM architectures.

Since gesture recognition is comparison between time series (in one or more dimensions), it could be interesting to experiment with methods for indexing data sets, as the gesture library is just such a data set. While such indexing techniques such as iSAX [31], or the multidimensional HyperSAX [11], are intended for very large data sets (millions), the concept of building and searching an index may prove to be faster than comparing the input gesture with each gesture trace in the library.

References

[1] Bakker, A. Comparing energy profilers for android. In *21st Twente Student Conference on IT* (2014).

[2] Baptista, R. Fast approximate distance functions, 2003. `http://www.flipcode.com/archives/Fast_Approximate_Distance_Functions.shtml`.

[3] Benini, L., De Micheli, G., et al. Energy-efficient design of battery-powered embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 9*, 1 (2001), 15–28.

[4] Berg Insight. Smart homes and home automation - 3rd edition, 2014. http://www.berginsight.com/ShowReport.aspx?m_m=3&id=201.

[5] Business Insider UK. The wearables report: Growth trends, consumer attitudes, and why smartwatches will dominate, 2015. http://uk.businessinsider.com/the-wearable-computing-market-report-2014-10.

[6] De Vogeleer, K., Memmi, G., Jouvelot, P., and Coelho, F. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 793–803.

[7] Desrochers, S., Paradis, C., and Weaver, V. M. Initial validation of dram and gpu rapl power measurements. Tech. rep., University of Maine, 2015.

[8] Field, H., Anderson, G., and Eder, K. EACOF: a framework for providing energy transparency to enable energy-aware software development. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (2014), ACM, pp. 1194–1199.

[9] Fog, A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus, January 2016. http://www.agner.org/optimize/instruction_tables.pdf.

[10] Ghasemzadeh, H., Amini, N., Saeedi, R., and Sarrafzadeh, M. Power-aware computing in wearable sensor networks: an optimal feature selection. *Mobile Computing, IEEE Transactions on 14*, 4 (2015), 800–812.

[11] Gydesen, J. E., Haxholm, H., Poulsen, N. S., Wahl, S., and Thiesson, B. Hypersax: Fast approximate search of multidimensional data. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods* (2015), pp. 190–198.

[12] Hähnel, M., Döbel, B., Völp, M., and Härtig, H. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review 40*, 3 (2012), 13–17.

[13] Hao, S., Li, D., Halfond, W. G., and Govindan, R. Estimating android applications' CPU energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software* (2012), IEEE Press, pp. 1–7.

[14] Herold, J., and Stahovich, T. F. The 1¢ recognizer: a fast, accurate, and easy-to-implement handwritten gesture recognition technique. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling* (2012), Eurographics Association, pp. 39–46.

[15] Hsieh, P. Programming optimization, 2016. http://www.azillionmonkeys.com/qed/optimize.html.

[16] Keogh, E. J., and Pazzani, M. J. Scaling up dynamic time warping for datamining applications. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining* (2000), ACM, pp. 285–289.

[17] Kerrison, S., and Eder, K. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems (TECS) 14*, 3 (2015), 56.

[18] Kowarschik, M., and Weiss, C. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*. Springer, 2003, pp. 213–232.

[19] Kratz, S., and Rohs, M. A $3 gesture recognizer: Simple gesture recognition for devices equipped with 3d acceleration sensors. In *Proceedings of the 15th International Conference on Intelligent User Interfaces* (New York, NY, USA, 2010), IUI '10, ACM, pp. 341–344.

[20] Lane, N., and Campbell, A. The influence of microprocessor instructions on the energy consumption of wireless sensor networks. In *Third Workshop on Embedded Networked Sensors (EmNets 2006)* (2006), vol. 34.

[21] Lee, S., Ermedahl, A., Min, S. L., and Chang, N. An accurate instruction-level energy consumption model for embedded risc processors. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 1–10.

[22] Milosevic, B., Farella, E., and Benini, L. Continuous gesture recognition for resource constrained smart objects. In *Proceedings of the fourth international conference on mobile ubiquitous computing, systems, services and technologies, UBICOMM* (2010), pp. 391–396.

[23] Naveh, A., Rajwan, D., Ananthakrishnan, A., and Weissmann, E. Power management architecture of the 2nd generation Intel® Core™ microarchitecture, formerly codenamed Sandy Bridge. In *Hot Chips* (2011), vol. 23, p. 0.

[24] Niezen, G., and Hancke, G. P. Evaluating and optimising accelerometer-based gesture recognition techniques for mobile devices. *The 9th IEEE AFRICON* (2009), 1–6.

[25] Nikolaidis, S., Kavvadias, N., Laopoulos, T., Bisdounis, L., and Blionas, S. Instruction level energy modeling for pipelined processors. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Springer, 2003, pp. 279–288.

[26] Nikolaidis, S., Kavvadias, N., and Neofotistos, P. Instruction level power measurements and analysis. *Aristotle University of Thessaloniki, Thessaloniki, Greece, Deliverable EASY/WP2/AUTH/DL/P D 15* (2002).

[27] Prekopcsák, Z. Accelerometer based real-time gesture recognition. (2008).

[28] Raffa, G., Lee, J., Nachman, L., and Song, J. Don't slow me down: Bringing energy efficiency to continuous gesture recognition. In *Wearable Computers (ISWC), 2010 International Symposium on* (2010), IEEE, pp. 1–8.

[29] Sakoe, H., and Chiba, S. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on 26*, 1 (1978), 43–49.

[30] Salvador, S., and Chan, P. Fastdtw: Toward accurate dynamic time. *Warping in Linear Time and Space* (2007).

[31] Shieh, J., and Keogh, E. isax: disk-aware mining and indexing of massive time series datasets. *Data Mining and Knowledge Discovery 19*, 1 (2009), 24–57.

[32] Šimunić, T., Benini, L., De Micheli, G., and Hans, M. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the 13th international symposium on System synthesis* (2000), IEEE Computer Society, pp. 193–198.

[33] Tiwari, V., Malik, S., and Wolfe, A. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 2*, 4 (1994), 437–445.

[34] Wang, S.-C. Artificial neural network. In *Interdisciplinary Computing in Java Programming*. Springer, 2003, pp. 81–100.

[35] Weste Neil, H., and Eshraghian, K. Principles of cmos vlsi design–a system perspective, 1993.

[36] Westeyn, T., Brashear, H., Atrash, A., and Starner, T. Georgia tech gesture toolkit: supporting experiments in gesture recognition. In *Proceedings of the 5th international conference on Multimodal interfaces* (2003), ACM, pp. 85–92.

[37] Wilke, C., Götz, S., and Richly, S. Jouleunit: a generic framework for software energy profiling and testing. In *Proceedings of the 2013 workshop on Green in/by software engineering* (2013), ACM, pp. 9–14.

[38] Wobbrock, J. O., Wilson, A. D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology* (2007), ACM, pp. 159–168.

[39] Wu, J., Pan, G., Zhang, D., Qi, G., and Li, S. Gesture recognition with a 3-d accelerometer. In *Ubiquitous intelligence and computing*. Springer, 2009, pp. 25–38.

[40] Yan, Z., Subbaraju, V., Chakraborty, D., Misra, A., and Aberer, K. Energy-efficient continuous activity recognition on mobile phones: An activity-adaptive approach. In *Wearable Computers (ISWC), 2012 16th International Symposium on* (2012), Ieee, pp. 17–24.

[41] Yuan, W., and Nahrstedt, K. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 149–163.

[42] Zinnen, A., and Schiele, B. A new approach to enable gesture recognition in continuous data streams. In *Wearable Computers, 2008. ISWC 2008. 12th IEEE International Symposium on* (2008), IEEE, pp. 33–40.

A Evaluation Results Statistics

Energy

Optimization	Average	Median	Min	Max	Standard Deviation
0	201.78 mJ	198.52 mJ	187.92 mJ	221.85 mJ	8.64
1	193.51 mJ	194.49 mJ	185.34 mJ	203.26 mJ	3.05
2	175.63 mJ	176.74 mJ	167.41 mJ	179.41 mJ	3.11
3	103.69 mJ	101.45 mJ	92.15 mJ	124.32 mJ	4.56
4	173.6 mJ	176.72 mJ	167.49 mJ	180.17 mJ	4.38
5	96.53 mJ	94.02 mJ	89.67 mJ	117.19 mJ	5.02

Time

Optimization	Average	Median	Min	Max	Standard Deviation
0	27.46 ms	27.51 ms	26.99 ms	29 ms	0.26
1	26.96 ms	27.05 ms	26.57 ms	27.49 ms	0.19
2	23.9 ms	23.97 ms	23.55 ms	24.09 ms	0.16
3	13.19 ms	13.28 ms	12.58 ms	13.56 ms	0.21
4	23.57 ms	23.63 ms	23.19 ms	24.53 ms	0.19
5	12.5 ms	12.59 ms	11.89 ms	13.2 ms	0.22

*Table 10: Segmentation measurement statistics with long trace (*100 min/1000 *gestures*

Energy

Optimization	Average	Median	Min	Max	Standard Deviation
0	133.29 J	131.66 J	131.46 J	168.47 J	5.75
6	116.57 J	116.33 J	116.25 J	117.42 J	0.34
7	110.31 J	109.95 J	109.86 J	114.49 J	0.6
15	10.1 J	10.09 J	10.08 J	10.32 J	0.02
16	128.62 J	128.07 J	127.97 J	148.23 J	2.14
17	127.38 J	127.71 J	126.22 J	136.03 J	1.03

Time

Optimization	Average	Median	Min	Max	Standard Deviation
0	18.32 s	18.14 s	18.12 s	23 s	0.79
6	16 s	16 s	15.98 s	16.02 s	0
7	15.13 s	15.12 s	15.1 s	15.6 s	0.05
15	1.39 s	1.39 s	1.39 s	1.39 s	0
16	17.69 s	17.65 s	17.65 s	20.27 s	0.27
17	17.52 s	17.51 s	17.49 s	18.54 s	0.1

*Table 11: DTW Gesture Recognizer measurement statistics with short trace (*10 min/100 *gestures)*

Energy

Optimization	Average	Median	Min	Max	Standard Deviation
0	1,160.52 mJ	1,160.22 mJ	1,149.87 mJ	1,176.86 mJ	5.77
6	1,161.53 mJ	1,160.23 mJ	1,154.9 mJ	1,170.79 mJ	3.73
7	1,124.51 mJ	1,124.26 mJ	1,114.61 mJ	1,134.13 mJ	4.11
8	1,150.12 mJ	1,149.34 mJ	1,142.18 mJ	1,174.39 mJ	5.74
9	1,105.64 mJ	1,106.65 mJ	1,098.33 mJ	1,126.69 mJ	4.35
10	1,157.48 mJ	1,157.18 mJ	1,149.97 mJ	1,180.57 mJ	5.49
11	1,110.79 mJ	1,111.01 mJ	1,102.98 mJ	1,133.16 mJ	4.13
12	1,116.48 mJ	1,115.81 mJ	1,108.12 mJ	1,143.78 mJ	4.93
13	1,103 mJ	1,101.66 mJ	1,097.95 mJ	1,125.72 mJ	3.78
14	1,148.39 mJ	1,147.9 mJ	1,137.73 mJ	1,183.18 mJ	5.66
15	273.04 mJ	272.75 mJ	267.49 mJ	279.27 mJ	3.19
16	890.84 mJ	890.48 mJ	883.03 mJ	928.84 mJ	5.41

Time

Optimization	Average	Median	Min	Max	Standard Deviation
0	158.4 ms	158.44 ms	157.91 ms	159.32 ms	0.25
6	159.31 ms	159.34 ms	158.72 ms	159.95 ms	0.23
7	153.41 ms	153.44 ms	152.69 ms	154.28 ms	0.27
8	157.79 ms	157.82 ms	156.89 ms	158.51 ms	0.29
9	148.57 ms	148.63 ms	148.04 ms	149.08 ms	0.24
10	158.76 ms	158.79 ms	158.17 ms	159.37 ms	0.26
11	148.81 ms	148.83 ms	148.33 ms	149.49 ms	0.24
12	152.96 ms	152.95 ms	152.01 ms	155.9 ms	0.4
13	147.27 ms	147.3 ms	146.76 ms	147.87 ms	0.22
14	156.49 ms	156.47 ms	155.95 ms	160.12 ms	0.44
15	38.17 ms	38.22 ms	37.71 ms	38.64 ms	0.19
16	122.5 ms	122.48 ms	121.95 ms	126.4 ms	0.46

*Table 12: 1¢ Gesture Recognizer measurement statistics with long trace (*100 min*/*1000 *gestures)*

B Distance Functions

B.1 One Dimensional

Let $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ be two one dimensional data series of length n. The distance between X and Y can be calculated using the following equations:

Euclidean

$$Dist(X,Y) = \sqrt{\sum_{i=0}^{n}(x_i - y_i)^2}$$

Squared Euclidean

$$Dist(X,Y) = \sum_{i=0}^{n}(x_i - y_i)^2$$

Taxicab

$$Dist(X,Y) = \sum_{i=0}^{n}|x_i - y_i|$$

B.2 Three Dimensional

Let

$$A = \{(a_{x_1}, a_{y_1}, a_{z_1}), (a_{x_2}, a_{y_2}, a_{z_2}), \ldots, (a_{x_n}, a_{y_n}, a_{z_n})\}$$

and

$$B = \{(b_{x_1}, b_{y_1}, b_{z_1}), (b_{x_2}, b_{y_2}, b_{z_2}), \ldots, (b_{x_n}, b_{y_n}, b_{z_n})\}$$

be two three dimensional data series of length n. The distance between A and B can be calculated using the following equations:

Euclidean

$$Dist(A,B) = \sqrt{\sum_{i=0}^{n}(a_{x_i} - b_{x_i})^2 + (a_{y_i} - b_{y_i})^2 + (a_{z_i} - b_{z_i})^2}$$

Squared Euclidean

$$Dist(A,B) = \sum_{i=0}^{n}(a_{x_i} - b_{x_i})^2 + (a_{y_i} - b_{y_i})^2 + (a_{z_i} - b_{z_i})^2$$

Taxicab

$$Dist(A,B) = \sum_{i=0}^{n}|a_{x_i} - b_{x_i}| + |a_{y_i} - b_{y_i}| + |a_{z_i} - b_{z_i}|$$

Octogonal [2]

$$Dist(A,B) = \sum_{i=0}^{n}\left(\frac{441}{1024} \times \min\begin{Bmatrix}|a_{x_i} - b_{x_i}|\\ |a_{y_i} - b_{y_i}|\\ |a_{z_i} - b_{z_i}|\end{Bmatrix} + \frac{1007}{1024} \times \max\begin{Bmatrix}|a_{x_i} - b_{x_i}|\\ |a_{y_i} - b_{y_i}|\\ |a_{z_i} - b_{z_i}|\end{Bmatrix}\right)$$

C 1¢ Pseudocode

```
function RESAMPLE(points, n)
    I ← PATH-LENGTH(points)/(n − 1)
    D ← 0
    newPoints ← points_0
    for all point p_i for i ≥ 1 in points do
        if D + d ≥ I then
            q_x ← p_{i−1_x} + ((I − D)/d) × (p_{i_x} − p_{i−1_x})
            q_y ← p_{i−1_y} + ((I − D)/d) × (p_{i_y} − p_{i−1_y})
            APPEND(newPoints, q)
            INSERT(points, i, q)
            D ← 0
        else
            D ← D + d
        end if
    end for
    return newPoints
end function

function PATH-LENGTH(A)
    d ← 0
    for i from 1 to |A| step 1 do
        d ← d + DISTANCE(A_{i−1}, A_i)
    end for
    return d
end function

function C-DISTANCE(points)
    c ← CENTROID(points)
    for all point p in points do
        d ← DISTANCE(c, p)
        APPEND(distances, d)
    end for
    return distances
end function

function Z-NORMALIZE(S)
    μ ← AVERAGE(S)
    σ ← STANDARD-DEVIATION(S)
    for all d in S do
        z ← (d − μ)/σ
        APPEND(z, d_z)
    end for
    return d_z
end function

function RECOGNIZE(S, Templates)
    for all template T in Templates do
        b ← ∞
        d ← L2(S, T)
        if d < b then
            b ← d
            T* ← T
        end if
    end for
    return T*
end function

function L2(S,T)
    d ← 0
    for all s_i, t_i for i ≥ 1 in S,T do
        d ← d + (s_i − t_i)^2
    end for
    return d
end function
```

Listing 5: Pseudocode for the 1¢ gesture recognizer. Copied from [14].

D Perf Counter Table

Difference (%)	DTW	Onecent	Counter
85,293	$4.11 \cdot 10^{12}$	$4.81 \cdot 10^{9}$	L1-dcache-loads
85,009	$1.06 \cdot 10^{13}$	$1.25 \cdot 10^{10}$	instructions
71,525	$1.67 \cdot 10^{12}$	$2.33 \cdot 10^{9}$	mem-stores
70,804	$1.67 \cdot 10^{12}$	$2.36 \cdot 10^{9}$	L1-dcache-stores
60,537	13,158.13	21.7	power/energy-cores/
58,982	$4.51 \cdot 10^{12}$	$7.63 \cdot 10^{9}$	cpu-cycles
58,950	$1.89 \cdot 10^{6}$	3,195.76	task-clock
49,510	3,909.25	7.88	power/energy-ram/
16,229	$2.15 \cdot 10^{8}$	$1.31 \cdot 10^{6}$	L1-icache-load-misses
8,141	$7.77 \cdot 10^{8}$	$9.43 \cdot 10^{6}$	LLC-stores
6,972	$6.81 \cdot 10^{9}$	$9.63 \cdot 10^{7}$	L1-dcache-load-misses
2,375	$4.24 \cdot 10^{9}$	$1.71 \cdot 10^{8}$	cache-references
834	62,132	6,654	page-faults
585	$1.87 \cdot 10^{8}$	$2.73 \cdot 10^{7}$	LLC-loads
3.99	$1.72 \cdot 10^{8}$	$1.66 \cdot 10^{8}$	cache-misses
0.84	$2.56 \cdot 10^{7}$	$2.53 \cdot 10^{7}$	LLC-load-misses
−12.17	$3.85 \cdot 10^{5}$	$4.38 \cdot 10^{5}$	LLC-store-misses

Table 13: Results from running `perf` on DTW and 1¢, and the difference between the results of the two algorithms.

E Intel Architectures

Architecture Name	Release Year	RAPL Support
Bonnell	2008	No
Sandy Bridge	2011	Yes, but no DRAM
Ivy Bridge	2012	Yes, but no DRAM
Haswell	2013	Yes
Broadwell	2014	Yes
Skylake	2015	Yes

Table 14: List of newer and released (by June 2016) Intel microarchitectures