Spring 4-30-2018

Detailed Power Measurement with Arm Embedded Boards

Yanxiang Mao
University of Maine, lanlongzhiwang@gmail.com

DETAILED POWER MEASUREMENT WITH ARM EMBEDDED BOARDS

By

Yanxiang Mao

B.S. University of Maine, 2015

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Engineering)

The Graduate School

The University of Maine

May 2018

Advisory Committee:

Vincent Weaver, Assistant Professor, Advisor

Richard Eason, Associate Professor

Bruce Segee, Henry R. and Grace V. Butler Professor

DETAILED POWER MEASUREMENT WITH ARM EMBEDDED BOARDS

By Yanxiang Mao

Thesis Advisor: Vincent Weaver

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Engineering)

May 2018

Power and energy are becoming important considerations in today's electronic equipment. The amount of power required to run a supercomputer for an hour could supply an ordinary household for many months. The need for low-power computing also extends to smaller devices, such as mobile phones, laptops and embedded devices.

In order to optimize power usage of electronic equipment, we need to collect information on the power consumption of these devices. Unfortunately it is not easy to do this on modern computing systems. Existing measuring equipment is often expensive, inaccurate, and difficult to operate. The main goal of this project is to create low-cost, accurate, stable, and easy-to-operate measurement equipment.

We design a low-cost low-overhead power measurement device using a Teensy embedded board. We then test our device by measuring the power consumption of a Raspberry Pi embedded board under a variety of different workloads. We compare the results with existing measurement equipment and analyze the advantages and shortcomings of our design.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
INTRODUCTION AND MOTIVATION

In the design of modern electronic equipment, energy and power consumption have become increasingly important. In order to design better products, engineers need to understand the power behavior of previous products, which they can use to roughly estimate the performance of new products and future energy costs. It is not easy to measure power data. There are a variety of issues with existing measuring equipment, including high initial cost, data inaccuracies, low sampling rate and high operational difficulty.

The goal of this research is to design a low cost, stable, high resolution, high sampling rate, and easy-to-use measurement device to collect power consumption data. These data should be trustworthy and easy to analyze.

1.1 Motivation

Our research group wanted to obtain high-sample rate power measurements and found this was much more difficult than imagined. Our explorations of this area show that much work can be done in this field. The research has wider impacts than just gathering power measurements for our research group. In terms of commercial design, power savings are always a good thing, from large-scale devices down to small-scale embedded systems. A typical large-scale computing environment is a supercomputer. In the Top 500 supercomputer list as of June 2017, the No. 1 Sunway TaihuLight has 10,649,600 cores and uses 15,371 kW of power [2]. Powering this supercomputer for a day uses electricity roughly equivalent to a month of electricity consumption for 700 Bangor, Maine, area people [1]. Even if we can only shave 0.1 Watts of power off of each processor core of this supercomputer, the total power savings would be significant.

Mobile phones are typical of modern embedded systems; power saving means that the phone can run for longer. Who does not want their cell phone to last longer after each charge? Saving power is not always easy. As System on Chip (SoC) technology advances, saving energy from the hardware becomes more and more difficult. Many companies have given up on saving energy from the hardware and try to find a way from software. Saving energy also reduces heat, fan noise, and provides other benefits. In order to optimize software to consume less power, there has to be some way to quantify the power usage. Our goal is to provide ways to gather this information.

1.2 Background

Power and energy are two commonly used physical properties. They represent different things, but usually can be transformed into each other.

1.2.1 Power

In physics, power is the rate of doing work. It is the amount of energy consumed per unit time. Electric power is the rate, per unit time, at which electrical energy is transferred by an electric circuit. The International System of Units (SI unit) of power is the Watt, one Joule per second. Power is very important in the circuit, not enough power will stop the circuit, while too much power will overload the circuit.

The electric power in Watts produced by an electric current of I Amperes consisting of a charge of Q Coulombs every t seconds passing through an electric potential (voltage) difference of V Volts [7] can be found by:

$$P = \frac{V * Q}{t} = V * I = I^2 * R = \frac{V^2}{R}$$

where Q is electric charge in Coulombs, t is time in seconds, I is electric current in Amperes, and V is electric potential (or voltage) in Volts.

In the acquisition of electricity data, sometimes it is difficult to know the exact resistance of the system, so it is best to use the form of this formula without resistance to calculate the power.

1.2.2 Energy

In physics, energy is the property that must be transferred to an object in order to perform work on or to heat the object. Electrical energy is the energy newly derived from electric potential energy or kinetic energy. The SI unit for energy is the Joule, but the energy unit used for everyday electricity, particularly for utility bills, is the kilowatt-hour (kWh); one kWh is equivalent to 3.6×10^6J (3600 kJ or 3.6 MJ) [7].

Energy can be used to measure the total electricity consumption of a device. The relationship between energy and power conversion is as follows:

$$E = P * t$$

1.3 Measurement

In the actual measurement of devices, we mainly focus on power. Power is a key metric with performance optimization because measuring energy requires the additional measurement of time, which is more complex. Energy is also important, but energy and power can be converted to each other. By optimizing for power, energy can also be optimized.

In order to get more accurate reference data we try to keep the conditions in the experiment simple. We could face various problems during measurements, including power supply instability, unknown external device power consumption, and software interference. It is almost impossible to completely avoid the interference of these factors in the actual measurement, but we can gradually weaken the influence of

these factors by constantly tracking them. Increasing the sampling rate and accuracy can help characterize the influence of these external factors.

Chapter 2
RELATED WORK

Power measurement is difficult, but it is necessary for many types of analysis. There is much previous work with regards to power measurement.

2.1 Estimated Power

Instead of measuring power, it is possible to estimate power usage by creating models based on various parameters, such as performance counter data, temperature, etc. [26, 9].

An extension of this is the Running Average Power Limit (RAPL) measurements found in recent Intel processors [11, 24]. These are typically estimated power values that are provided by a helper processor and can be read by the user. While useful, we prefer to do actual power measurement. Also, RAPL readings are only available on Intel systems, and not available on ARM based platforms.

2.2 Commercial Devices

You can buy devices that can measure power at the wall outlet, such as the WattsUpPro? power meter [15]. These tend to be expensive (over $100) and only have a resolution of 1Hz. Also you cannot get detailed power measurement, only total system power.

Hackenberg et al. [19] validate RAPL power measurements in a server, but use multi-thousand dollar calibrated voltmeters.

Desrochers et al. [12] measure detailed DRAM memory power consumption, but use a commercial digital/analog data acquisition device which is much more expensive that the device we propose.

2.3 Low-Cost Power Measurement

Paradis [21] looked into using Teensy boards to measure performance of server-class workloads. Our goal is to extend this but to also measure embedded systems.

Others have tried taking extremely small embedded devices and creating small data acquisition devices suitable for power measurement, such as the IViny based on the ATtiny85 [5].

Schaff and Weaver [25] investigate the accuracy of an Adafruit USB Power Gauge with an ATtiny85 and INA169 current sensor. They find that the A/D converter on the ATtiny85 is not as accurate as using a dedicated volt meter to measure the results from the current sensor.

Moreno et al. [20] use the sound card of a desktop system as a low-cost A/D converter for obtaining low-cost power measurements.

Bedard et al. [8] propose PowerMon2, a device that measures power inside of a server system.

Ge et al. [16] instrument an entire cluster for fine-grained power measurement, but this involved intrusive instrumentation of each machine.

Rashti et al. [23] design a PCIe card capable of measuring server power. We wish to measure embedded systems power though.

2.4 Raspberry Pi Power Measurement

Other researchers have measured the power of Raspberry Pi systems.

Geerling [17] has done a similar experiment with the Dramble cluster. They measure the Raspberry Pi power, and note that one must be sure to turn off the HDMI interface and LEDs to save power. Unfortunately full experimental details are not given.

Cloutier et al. [10] instrument a Raspberry Pi cluster and allow measurement at a per-node level using SPI A/D converters. This is not a general purpose solution though. They also provide power measurements of Raspberry Pis, but only for the Linpack benchmark.

On the Raspberry Pi Forum, Eames designed a similar experiment [13]. However, the experimental process is not rigorous, and the experimenter did not finish the whole experiment completely. Eames also measured the Raspberry Pi 2's power [14] relatively systematically. We carry out similar experiments. This was done because there is no comprehensive official results for how much power a Raspberry Pi consumes.

Chapter 3
EXPERIMENTAL SETUP

3.1 Hardware

This section will talk about the hardware component of our power measurement setup. A diagram of our test setup is shown in Figure 3.1.

3.1.1 Teensy

The core part of our measurement device is a Teensy development board, shown in Figure 3.2. The Teensy is used to collect data and to upload this data to the computer for subsequent analysis. We use a Teensy model 3.1 in this work. This model has a MK20DX256 32 bit ARM Cortex-M4 72 MHz processor, one 12-bit resolution ADC with 18 MHz frequency and 10 easy to use analog pins [6]. The Teensy can be operated from 3.6 to 6.0 volts. Also the Teensy has a 3.3V (100 mA max) internal power supply. Unfortunately, the Teensy 3.1 is currently discontinued.

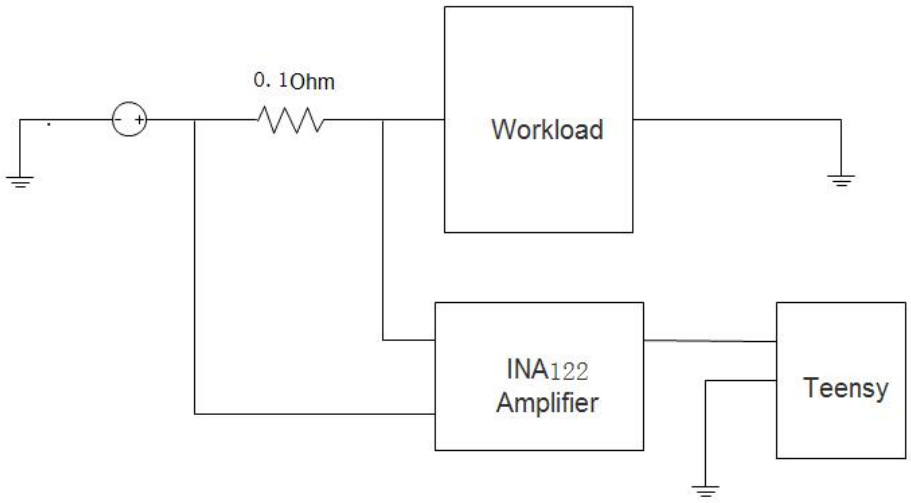

Figure 3.1: Our test setup.

Figure 3.2: Teensy 3.1 USB development board.

The lowest model currently available is Teensy 3.2, priced at $19.80. The main difference between the Teensy 3.1 and 3.2 is that the Teensy 3.2 adds a more powerful 3.3 volt regulator, with the ability to directly power ESP8266 Wifi, WIZ820io Ethernet and other power-hungry 3.3V add-on boards. These features should not affect this experiment, so it should be possible to reproduce this experiment with Teensy 3.2 without any problems.

3.1.2 INA122 Instrumentation Amplifier

The Teensy is a low-end embedded board, so while it has an analog to digital converter, it is not capable of measuring the small voltages required in our work. To reduce the error, we use an instrumentation amplifier. The INA122 instrumentation amplifier can provide gain proportional to a gain resistor R_G as shown in Figure 3.3:

$$G = 5 + \frac{200k}{R_G}$$

The voltages we are measuring are roughly from 10mV to 50mV, and the Teensy cannot measure voltages above 3.3V, so we increase the voltage range to fall between 1V and 2V. We chose a R_G of 7.5k, made by putting a 4.2k Ohm resistor and a 3.3k Ohm resistor in series. Their actual resistance is 4.22k Ohms and 3.25k Ohms, so by using the formula, we get an actual gain of $5 + 200k/7.47k = 31.77$.

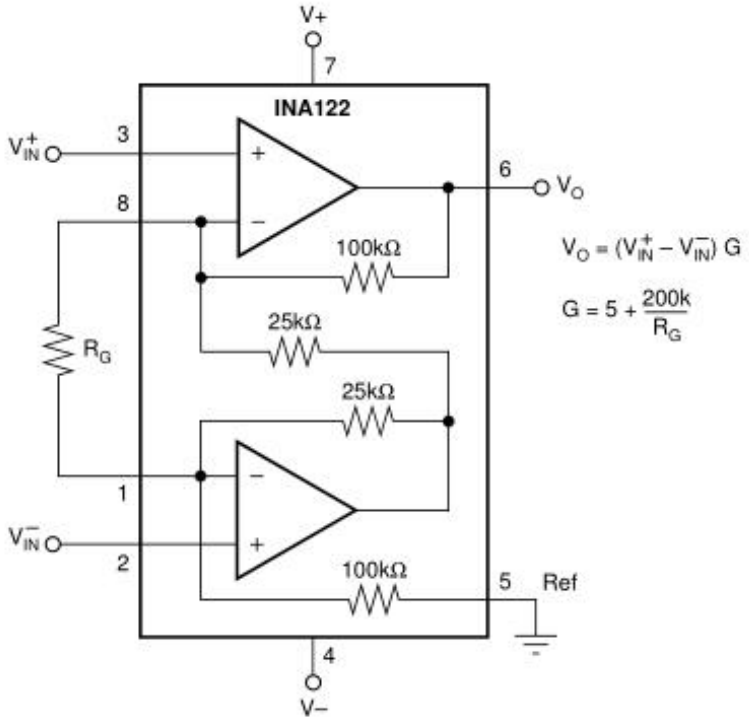

Figure 3.3: INA122 Instrumentation Amplifier block diagram

3.1.3 Voltmeter

To validate our measurements we need a reference. For this purpose we use a model Tenma 72–7770 multimeter as shown in Figure 3.4. The detailed values for the resistors in Section 3.1.2 are also measured by this multimeter.

3.1.4 Self-made power supply cable

For our experiments we will be measuring the power usage of a Raspberry Pi. To measure the power, we will need to measure the current draw across a sense resistor. The Pi does not have such a resistor built in, so we need to add a resistor. We built a custom cable that lets us insert a 0.1 Ohm resistor in series with the 5V power supply entering the Pi, as shown in Figure 3.5.

Figure 3.4: Tenma 72–7770 Voltmeter

3.2 Source code and Software Tool

This section describes the various programs used for measurement, as well as the core logic code for the Teensy. Some of the software failed for a variety of reasons, as will be discussed.

3.2.1 Operating System and Virtual Machine Setup

A virtual machine solution was used because development and experiments alternated between Windows and Linux. The primary system was Windows, so a Linux virtual machine was run under Oracle VirtualBox. The version of VirtualBox used was 5.1.28. In order to share the serial port with Windows, the extended package and VBOXADDITIONS was also installed. Linux is running Ubuntu Mate 64-bit version, the kernel version is 3.19.0-49-generic.

3.2.2 Teensyduino Environment

Teensyduino is a development environment based on Arduino, so before installing Teensy, we have to install Arduino first. In Windows, this is straightforward, just go

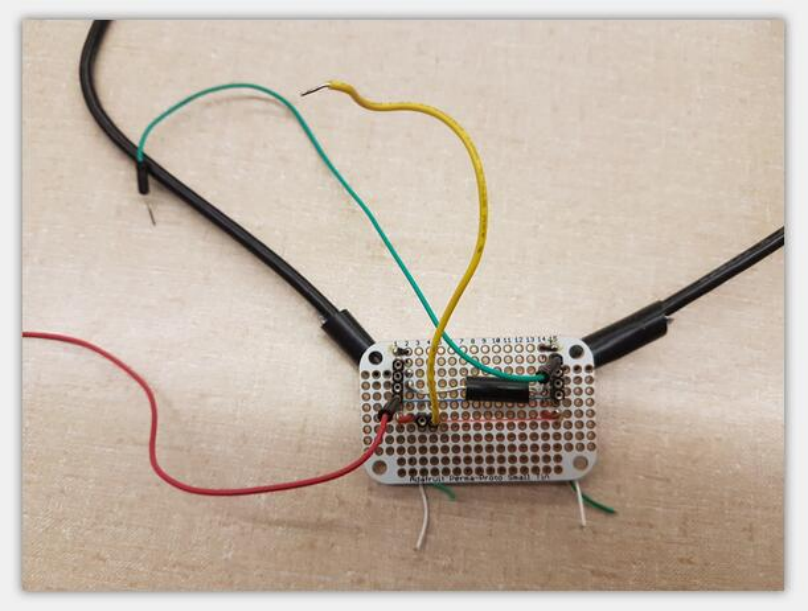

Figure 3.5: Custom-built power supply cable

to the Arduino website, download the program and install it. Then download the Teensyduino Windows Installer and install that. My main problem with the Windows version of Teensyduino is that Windows can not identify Teensy from the serial port. The Windows Serial Installer is available on Teensy's website, but sometimes it does not work. Windows can download the program to the Teensy, but the Teensy can not return the data. Three different computers were tried, all of them running on the Windows 7 operating system: one worked, the other two did not. Online it was found that someone solved this problem successfully by modifying the registry, but that did not work on either of these two computers.

On Linux, we also have to install Arduino first. When installing using `apt-get`, the Teensyduino Linux Installer will not work, so we need to note that the Teensyduino Linux Installer will only work for personal installation. Also under Linux we have to install udev rules.

The Teensyduino version used is 1.24, the newest version at the time of writing is 1.4 The more detailed differences between these two versions is discussed in Section 3.2.4.

3.2.3 Linux Perf Tool

The Linux perf tool [18] is used for performance measurement. It enables applications to take advantage of the hardware performance measurement unit (PMU), tracepoint, and special counters in the kernel to gather performance data. Not only does it analyze the per-thread performance of a given application, it can also be used to analyze performance issues with the kernel. This can be used to find performance bottlenecks.

Typing `sudo apt-get install linux-perf-4.4` in a Linux terminal will automatically install the perf tool. To use perf, one runs `perf stat [program]`. This will record statistics and provide an overview and performance summary of the program being debugged in a concise and condensed manner.

3.2.4 Main Program

This section discusses the code for the Teensy that implements our interface.

3.2.4.1 Initialization

The first part of our code does the initialization, as shown in Figure 3.6. This sets the Teensy's ADC resolution, average value and sampling speed, conversion speed and reference voltage. It also sets the serial transmission speed and LED indicator. The resolution of a Teensy ADC can reach up to 16-bit, but this will cause greater memory pressure, so here the resolution is set to 12-bit. The conversion speed is set to high speed, which is the highest speed within specs for resolutions less than 16 bits. For the sampling speed, the higher impedance should reduce the sampling speed; in this experiment, the impedance is small, so we set the sampling speed to high speed. We also set the reference voltage to 3.3 volts, combined with 12-bit resolution, meaning that the read data is converted to the actual voltage using the formula $(3.3/4096) \times data$. Also note the commented out

```
//Set up all  before running
void setup()
{
  Serial.begin(38400);
  pinMode(ledPin, OUTPUT);
  adc->setAveraging(4);
  adc->setResolution(12);
  /*adc->setConversionSpeed(ADC_HIGH_SPEED);
  adc->setSamplingSpeed(ADC_HIGH_SPEED);
  adc->setReference(ADC_REF_3V3);*/

  //ADC_REF_3V3, ADC_REF_EXT or ADC_REF_1V2
  adc->setConversionSpeed(ADC_CONVERSION_SPEED::HIGH_SPEED);
  adc->setSamplingSpeed(ADC_SAMPLING_SPEED::HIGH_SPEED);
  //ADC_REF_3V3, ADC_REF_EXT or ADC_REF_1V2
  adc->setReference(ADC_REFERENCE::REF_3V3);
}
```

Figure 3.6: Main program initialization

```
t: time, how long you want sampling, default is 1000ms (1s)
c: channel, which ADC channel you want enable
(A0 on Teensy 3.1 is channel 1.)
(For example, input 13, equal to 0b00001101, that means enable A0, A2 and A3.)
f: frequency, setup the sampling frequency, default frequency is 1 Hz
d: detail, check current enable channel and sample time.
b: beginning to sample
```

Figure 3.7: User interface

section, which is the difference between the code in this experiment at the Teensyduino 1.24 and 1.4 versions.

3.2.4.2 User Interface

Our code provides a helpful interface, making it easier for users to use the program. An example of the interface is shown in Figure 3.7.

```
Please input request (input h for help):
```

Figure 3.8: User interface help

```
The current enable channel is channel 1
(A0 on Teensy 3.1 is channel 1.)
(For example, input 13, equal to 0b00001101, that means enable A0, A2 and A3.)
The current sampling freqency is 1 Hz
Base on current sampling frequency and current sampling time, you will get 1 samples
```

Figure 3.9: Check settings interface

```
for (i=0; i<how_many_array_per_channel; i++){
  for (j=0; j<1000; j++){
    for (k=0; i<how_many_channel; i++){
      data[k][j] = adc->analogRead(open_channel[k]);
      delay(delay_time);
    }
  }

  for (m=0; m<how_many_channel; m++) {
    if (Serial.available()) {
      Serial.print("Channel ");
      Serial.print(open_channel[m]+1);
      Serial.print(" :");
    }
```

Figure 3.10: Main Logic

If the user does not know how to use this program they can enter 'h' to display the help interface, shown in Figure 3.8.

The default behavior is to open ADC channel 1, the sampling frequency is 1 Hz, and the sampling time is 1 second. Users can change these values at any time, but they also can type 'd' to check what the existing values are. It should be noted that after each restart the settings on the Teensy will be restored to the default values. The "check" interface is shown in Figure 3.9.

3.2.4.3 Main Logic

This section describes the most important part of the program, the main logic, as shown in Figure 3.10.

```
for (m=0; m<how_many_channel; m++) {
  if (Serial.available()) {
    Serial.print("Channel ");
    Serial.print(open_channel[m]+1);
    Serial.print(" :");
  }

  for (n=0; n<1000; n++){
    if (Serial.available()) {
        Serial.print(data[m][n], DEC);
        Serial.print(", ");
        }
    }

  if (Serial.available()) {
    Serial.println(" ");
  }

    memset(data[m],0,1000*sizeof(int));
}
```

Figure 3.11: Data Transfer

This code will sequentially scan each open port, take one data element from each port, and repeat until each port has taken over 1000 elements. It will then return the data to the computer through the serial port. The data transfer code is shown in Figure 3.11.

The benefit of this code is that it can loop indefinitely, no matter how long it takes to sample and how much data it needs. The disadvantage is when the data is returned through the serial port, data will temporarily suspend the sampling of the

```
gcc -o sleep sleep.c
```

Figure 3.12: Compiling the sleep benchmark

```
sudo perf_4.4 stat ./sleep
```

Figure 3.13: Using the sleep benchmark

ADC. A three-minute benchmark will take about three minutes and 10 seconds, so that sampling timeliness was affected.

Obtaining the data in one transaction was tried, but the Teensy 3.1's "each" array can only hold roughly 3,200 data elements, and the Teensy will stop working if it exceeds that number. A "multi-dimensional" array can bring that number to around 14,400, but that figure was still far behind being able to support taking a three-minute benchmark at a 1kHz sample rate. We made the trade-off of taking 1000 elements at time, while incurring some data loss. This allows the program to run stably over a long time period.

During the sampling process, an LED indicator is lit so the user can tell whether sampling is still underway.

3.3 Benchmarks

Various benchmarks were used to test the power measurement capabilities of the device. Namely Sleep, Bzip2, Iozone and Linpack.

3.3.1 Sleep

This benchmark was created for this work, using the C language `sleep()` function which tells the operating system to sleep (and potentially go into a low-power energy saving mode). This was used to test the Raspberry Pi while in low-power idle mode.

```
sudo perf_4.4 stat -e instructions:u bzip2 -k -f ./input.source
```

Figure 3.14: Using the Bzip2 benchmark

```
wget http://www.iozone.org/src/current/iozone3_471.tar
tar -xf iozone3_471.tar
cd iozone/iozone3_465/src/
```

Figure 3.15: Iozone installation.

Figure 3.12 shows how to compile the sleep benchmark. Figure 3.13 shows how to run the benchmark in the terminal while using the perf tool to monitor the operation of the program.

3.3.2 Bzip2

Bzip2 is a lossless compression software by Julian Seward based on the Burrows-Wheeler algorithm. It compresses better than the traditional LZ77 / LZ78 compression algorithms [3]. It is free software and is widely found in many distributions of UNIX and Linux. It can compress normal data files by 10% to 15%, and compression rates and decompression efficiencies are high.

Figure 3.14 shows how to use the perf tool to monitor the operation of this program.

3.3.3 Iozone

Iozone [4] is a file system benchmark tool. It can test the read and write performance of the file system in different operating systems. It can test the performance of hard disks in different modes such as read, write, re-read, re-write,

```
sudo perf_4.4 stat ./iozone -a -g 16M -i 0 -i 1
```

Figure 3.16: Using the Iozone benchmark.

```
sudo apt-get install libatlas-base-dev libmpich2-dev gfortran
wget http://www.netlib.org/benchmark/hpl/hpl-2.1.tar.gz
tar -xf hpl-2.1.tar.gz
cd hpl-2.1/setup
sh make_generic
cd ..
cp setup/Make.UNKNOWN Make.rpi
nano Make.rpi
"Modify the file as shown below"
ARCH         = rpi
TOPdir       = $(HOME)/hpl-2.1
MPdir        = /usr/local/mpich2
MPinc        = -I $(MPdir)/include
MPlib        = $(MPdir)/lib/libmpich.a
LAdir        = /usr/lib/atlas-base/
LAlib        = $(LAdir)/libf77blas.a $(LAdir)/libatlas.a
"ctrl+x"
make arch=rpi
```

Figure 3.17: HPL install attempt.

```
wget http://www.netlib.org/benchmark/linpackc.new
gcc -o linpack linpack.c -lm
```

Figure 3.18: Linpack100 installation.

read backwards, read strided, fread, fwrite, random read, pread, mmap, aio_read, aio_write.

Iozone was installed following the steps in Figure 3.15.

Figure 3.16 shows how we run iozone in our experiments. The maximum test filesize is 16M.

3.3.4 Linpack

Linpack [22] is internationally known as a popular benchmark for testing floating-point performance of high-Performance computer systems. It involves using

```
sudo perf_4.4 stat ./linpack
```

Figure 3.19: Using Linpack100.

Gaussian elimination to solve the first-order N-ary linear algebraic equations, exercising the floating-point performance of the system being evaluated.

Linpack tests include three categories, Linpack100, Linpack1000 and High Performance Linpack (HPL). HPL is used for supercomputer benchmarking and ranking on the TOP500 list [2]. Although it is possible to run HPL on the Raspberry Pi, I was personally unable to get it working. Details on the attempt are shown in Figure 3.17 for reference. Instead the Linpack100 benchmark was used. Linpack100 solves a dense linear algebraic equation of order 100, which allows optimization with the compiler optimizations only, with no changes to the code. The installation method is shown in Figure 3.18.

Figure 3.19 shows how to use the perf tool to monitor the operation of this program, including inputting the algebraic equation and order number.

Chapter 4
RESULTS AND DISCUSSION

This chapter contains the experimental results and analysis of the differences between the expected values and the experimental results. Static (system idle) and dynamic (system active) benchmarks are compared as well as results between a Raspberry Pi 2 and Raspberry Pi B.

4.1 Results

In this section the experimental results are shown in two categories: static and dynamic.

Static refers to the Raspberry Pi in idle mode. With the system idle, external devices such as keyboards, monitors, and network cables were gradually added, with the increase in power consumption noted. Since the system is idle, the voltage measured is relatively stable. We confirmed the data accuracy of the static data by comparing against voltmeter readings. It should be noted that the voltage of static data is only relatively stable, it still fluctuates, especially after connecting a network cable.

Dynamic data refers to the operation while running benchmarks. Under this kind of load the voltage has large fluctuations, making it harder to validate against the voltage meter readings. We tried to use the perf tool for data reference, but unfortunately the Raspberry Pi does not have a built-in power meter that can be read using this interface.

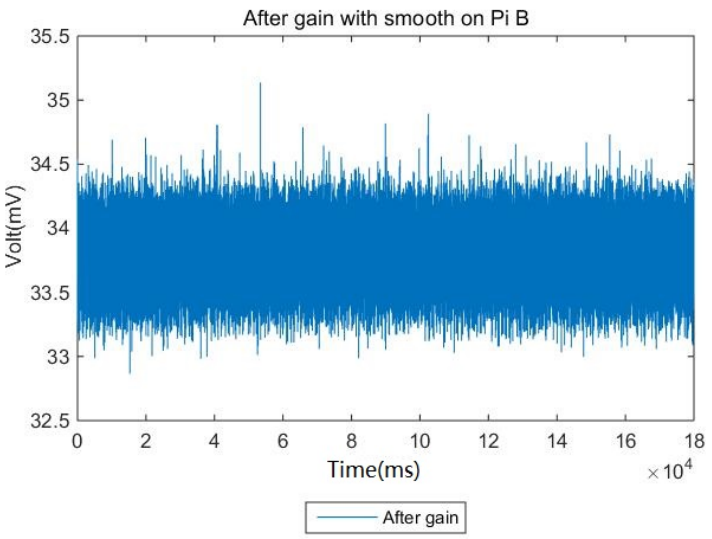

Figure 4.1: Raspberry Pi B Idle. The Y axis ranges from 1.625W to 1.775W

4.1.1 Raspberry Pi Power Measurement

Figure 4.1 shows the initial static data for the Raspberry Pi B. The amplified measurement is 1.121mV, with a maximum of 35.3mV across the sense resistor (equal to a power of 1.765W).

4.1.2 External Devices

In this section, we show the data obtained after the Raspberry Pi is connected to external devices. The purpose of this is to test whether the Teensy can measure the increased power consumption of these external devices. The data are: keyboard + Raspberry Pi, keyboard + display + Raspberry Pi, and Ethernet + Raspberry Pi. The above data sampling is performed on Raspberry Pi 2 and Raspberry Pi respectively, and a total of six groups of data are obtained, which are shown in Figure 4.2 and Figure 4.3.

First of all, in Figure 4.2, one can clearly see that the voltage increases with the addition of external devices. There is one problem to be aware of, that some of the readings were significantly higher during the second half of the test. I suspect that

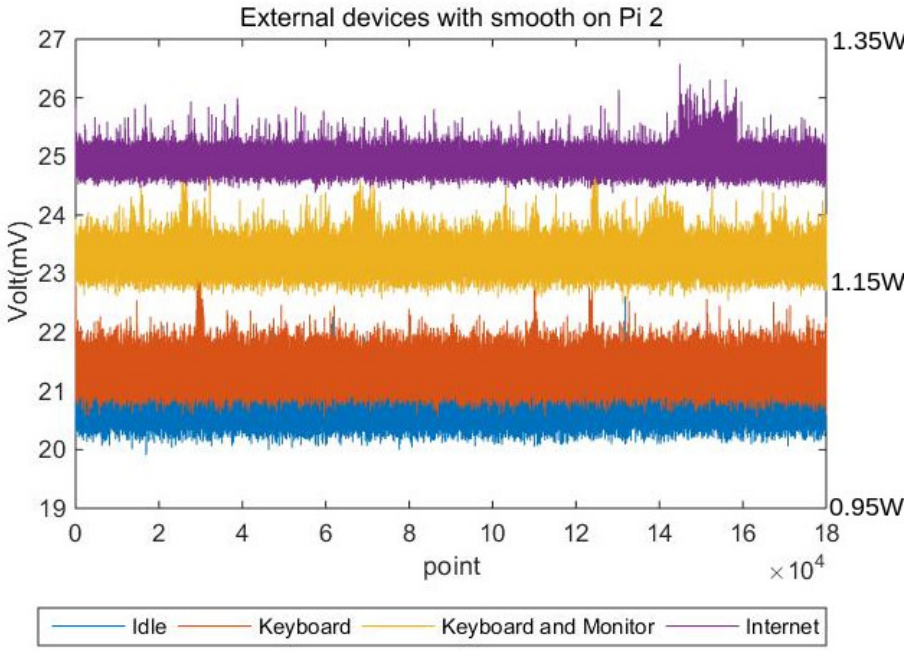

Figure 4.2: Raspberry Pi 2 External Devices

the Raspberry Pi 2 exchanged data over Ethernet in this short period of time, resulting in a brief rise in energy consumption. We found that the power consumption of Ethernet was significantly higher than that of the keyboard + display.

Figure 4.3 shows the Raspberry Pi B data. We cannot explain why for the second half of the keyboard + monitor measurements the results were lower than at other times.

Comparison of results between the Teensy and voltmeter are shown in Table 4.1. The error in Table 4.1 is relatively small, which should indicate that the Teensy will be relatively accurate when it comes to measuring higher voltage data.

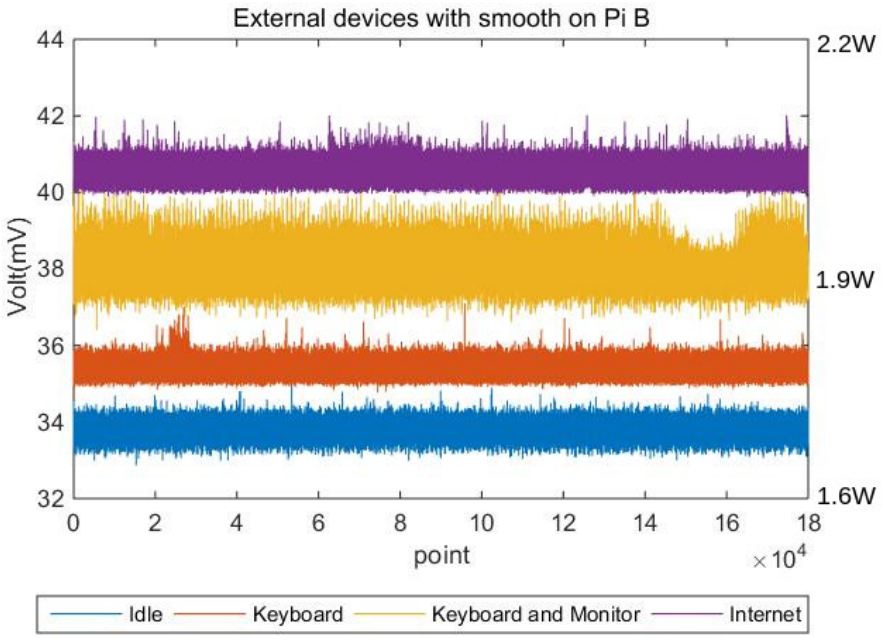

Figure 4.3: Raspberry Pi B External Devices.

Table 4.1: Pi measurement error.

	Voltmeter (mV)	Teensy(mV)	Error(%)
Raspberry Pi 2 Keyborard	21.2	21.32	0.5660377358
Raspberry Pi B Keyborad	35.3	35.35	0.1416430595
Raspberry Pi 2 Keyborard+Monitor	23	23.27	1.173913043
Raspberry Pi B Keyborard+Monitor	37.8	37.92	0.3174603175
Raspberry Pi 2 Internet	24.8	24.93	0.5241935484
Raspberry Pi B Internt	41	40.76	0.5853658537

4.2 Benchmark Results

4.2.1 Sleep

The sleep benchmark runs the C `sleep()` function for 170 seconds. This benchmark puts the Pi into idle mode, although some daemons and other system activity may still be running in the background and this can cause fluctuations. Figure 4.4 shows the power results, and Figure 4.6 shows the perf results.

One thing to note in Figure 4.6, we can see that this benchmark ran for 170 seconds. In Figure 4.4 the area with relatively large noise lasted for less than 140 seconds at a sampling frequency of 1000 Hz. The Sleep equation ran for less than 140 seconds and Perf showed that the program ran for 170 seconds and it is unclear why. find the possible explanation.

The Teensy data and perf tool data for Raspberry Pi B are shown in Figures 4.5 and 4.7. We can see a similar result to the Pi2 results. The difference is that there are more outliers in the sleep area. This may indicate that the Raspberry Pi was running other programs simultaneously while executing the `sleep()` function.

4.2.2 Bzip2 Results

Bzip2 is a compression benchmark included as part of the SPEC CPU2006 Benchmark suite [27]. SPEC's version of Bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem. The results of the Raspberry Pi 2 Teensy data and Perf tools are shown in Figures 4.8 and 4.10.

In Figure 4.8 we can see an area of significantly higher reading than the rest. In the Perf tool, it shows that the Bzip2 benchmark ran for 45.7 seconds. and we see the region did continue for roughly the same amount of time, indicating that Teensy did observe the Bzip2 benchmark. However, in this region, the readings are very messy and dense, and we can not determine the accuracy of these readings.

Figure 4.9 shows the Teensy results on Raspberry Pi B. Not only do these figures clearly show the approximate time of the Bzip2 benchmark, but there are also significant fluctuations in the curve. The perf results in Figure 4.11 are incomplete because the ARM1176 chip in the PiB is not capable of splitting out user and kernel mode events separately.

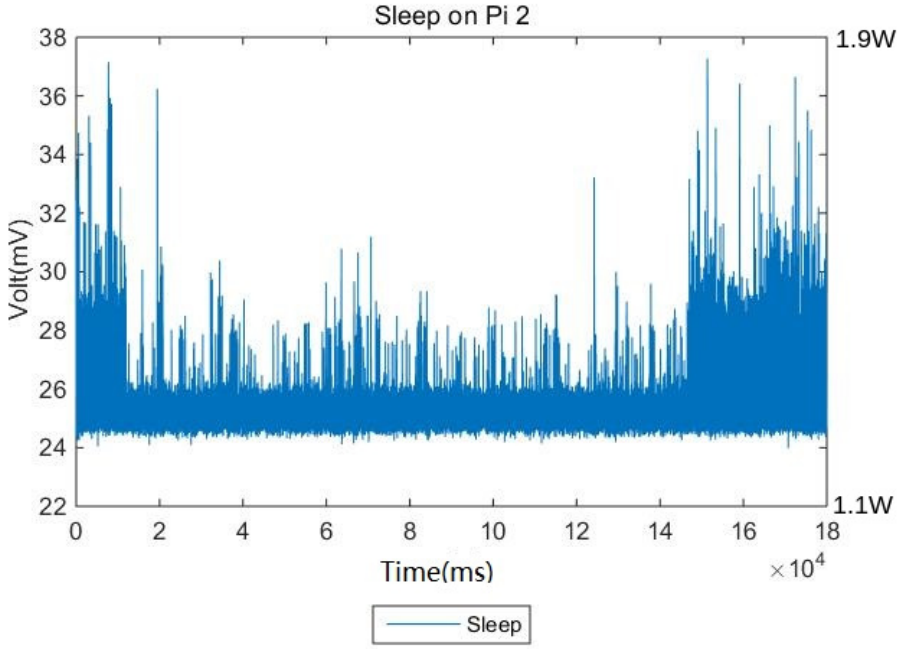

Figure 4.4: Sleep benchmark on Pi 2

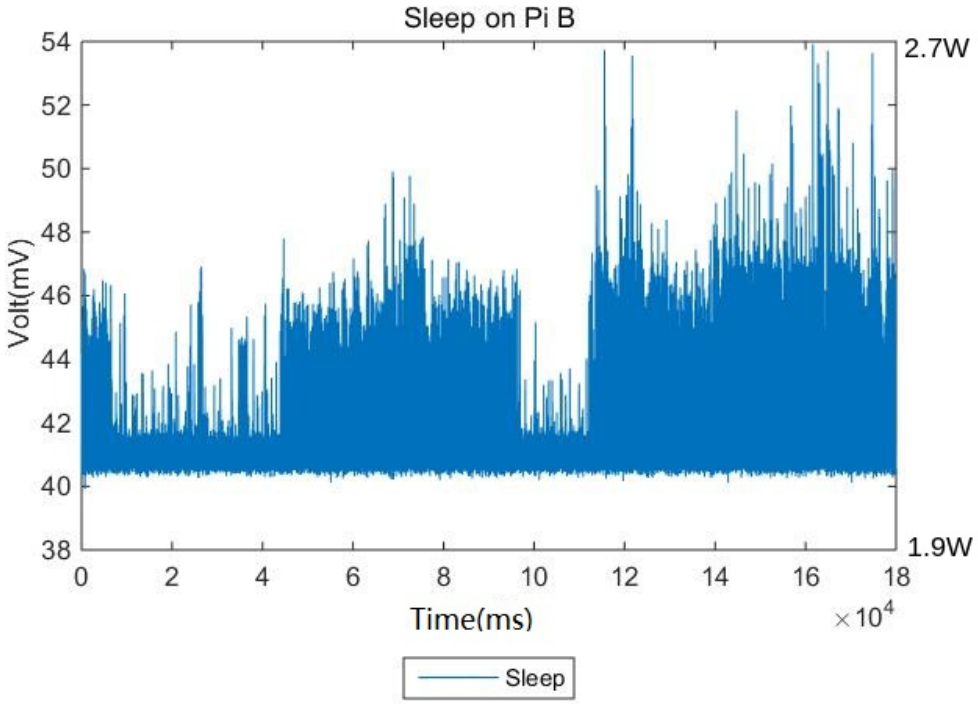

Figure 4.5: Sleep benchmark on Pi B

```
Performance counter stats for './sleep':

          3.267239      task-clock (msec)         #    0.000 CPUs utilized
                 1      context-switches          #    0.306 K/sec
                 0      cpu-migrations            #    0.000 K/sec
                39      page-faults               #    0.012 M/sec
         1,943,422      cycles                    #    0.595 GHz
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
           501,539      instructions              #    0.26  insns per cycle
            51,405      branches                  #   15.733 M/sec
            13,708      branch-misses             #   26.67% of all branches

     170.008078233 seconds time elapsed
```

Figure 4.6: Sleep perf results on Pi 2

```
Performance counter stats for './sleep':

          5.503000      task-clock (msec)         #    0.000 CPUs utilized
                 2      context-switches          #    0.363 K/sec
                 0      cpu-migrations            #    0.000 K/sec
                38      page-faults               #    0.007 M/sec
         1,273,167      cycles                    #    0.231 GHz
         2,310,511      stalled-cycles-frontend   #  181.48% frontend cycles idl
e
           163,654      stalled-cycles-backend    #   12.85% backend  cycles idl
e
     <not counted>      instructions              (0.00%)
     <not counted>      branches                  (0.00%)
     <not counted>      branch-misses             (0.00%)

     170.012376870 seconds time elapsed
```

Figure 4.7: Sleep perf results on Pi B

By comparing the Pi2 and PiB results, we can conclude that Teensy can indeed observe dynamic data. However, because of noise, we can not tell if the Teensy's reading is correct when the data changes rapidly.

4.2.3 Iozone Results

Iozone is a benchmark for stress testing file I/O. Figures 4.12 and 4.13 show Teensy data for Raspberry Pi 2 and Raspberry Pi B. Figures 4.14 and 4.15 are the results of the Perf tool.

This data shows disk I/O (SD card in the case of the Raspberry Pi) and should not be stressing the CPU or memory much.

4.2.4 Linpack100 Results

Linpack is a benchmark for testing floating-point performance of high-performance computer systems. Figures 4.16 and 4.17 show the results on Pi2 and PiB. The perf results are shown in Figures 4.18 and 4.19.

The use of floating point makes the power usage rise rapidly. The smaller fluctuations as the program progresses are hard to make out.

4.3 Power and Energy

In this section we will show the energy and power used by Raspberry Pi 2 and Raspberry Pi B for the benchmarks. It should be noted that for static data, the value is relatively stable, so the results will be more accurate. However, dynamic data has more variation and it can be harder to find the benchmark start and stop points.

Table 4.2 shows the data from idle mode. The result we should see is Raspberry Pi B power and energy consumption is much higher than Raspberry Pi 2. This is likely due to the more advanced power regulation circuitry introduced starting with the Pi B+ model. For comparison, Eames [14] reports Pi 2 current in Idle mode is

Table 4.2: Idle Energy

	Time (s)	Average Voltage (V)	Average Current (A)	Average Power (W)	Total Energy (J)
Raspberry Pi 2	180	0.02096	0.2096	1.048	188.64
Raspberry Pi B	180	0.03373	0.3373	1.6865	303.57

Table 4.3: Energy used by external devices

	Time (s)	Average Voltage (V)	Average Current (A)	Average Power (W)	Total Energy (J)
Raspberry Pi 2 Keyborard	180	0.02132	0.2132	1.066	191.88
Raspberry Pi B Keyborard	180	0.03535	0.3535	1.7675	318.15
Raspberry Pi 2 Keyborard+Monitor	180	0.02327	0.2327	1.1635	209.43
Raspberry Pi B Keyborard+Monitor	180	0.03792	0.3792	1.896	341.28
Raspberry Pi 2 Internet	180	0.02493	0.2493	1.2465	224.37
Raspberry Pi B Internet	180	0.04076	0.4076	2.038	366.84

Table 4.4: Sleep Energy

	Time (s)	Average Voltage (V)	Average Current (A)	Average Power (W)	Total Energy (J)
Raspberry Pi 2	180	0.02523	0.2523	1.2615	227.07
Raspberry Pi B	180	0.04105	0.4105	2.0525	369.45

Table 4.5: Bzip2 Energy

	Time (s)	Average Voltage (V)	Average Current (A)	Average Power (W)	Total Energy (J)
Raspberry Pi 2	45.72	0.0274	0.274	1.37	62.6364
Raspberry Pi B	106.64	0.0442	0.442	2.21	235.6744

Table 4.6: Iozone Energy

	Time (s)	Average Voltage (V)	Average Current (A)	Average Power (W)	Total Energy (J)
Raspberry Pi 2	100.1	0.0278	0.278	1.39	139.139
Raspberry Pi B	109.75	0.0442	0.442	2.21	242.5475

Table 4.7: Linpack100 Energy

	Time (s)	Average Voltage (V)	Average Current (A)	Average Power (W)	Total Energy (J)
Raspberry Pi 2	38.74	0.026	0.26	1.3	50.362
Raspberry Pi B	56.6	0.0417	0.417	2.085	118.011

0.230A while that of Pi B current is 0.360A. Our data is 0.2096A and 0.3373A, respectively which is very close to the figures given.

Next is the data for the external devices, shown in Table 4.3. With the Raspberry Pi 2, the keyboard power is 0.018W, HDMI interface power is 0.081W, and Ethernet power is 0.1985W. On Raspberry Pi B, the power of the keyboard is 0.081W, the power of the HDMI interface is 0.1285W, and the power of the Ethernet is 0.3515W. It can be seen that Raspberry Pi 2 has a significantly lower power consumption in all aspects than Raspberry Pi B.

Tables 4.4, 4.5, 4.6 and 4.7 show the power and energy consumption data for the four benchmarks Sleep, Bzip2, Iozone, and Linpack.

Because it is not possible to determine the end time of the Sleep benchmark, the average power and total energy consumption can only be calculated using a total length of 180 seconds, and it is possible to consume less energy during Sleep benchmarks than in Idle mode.

Among the four benchmarks above, Iozone has the highest energy consumption and Linpack has the lowest energy consumption. This is primarily due to Linpack100 having a short runtime.

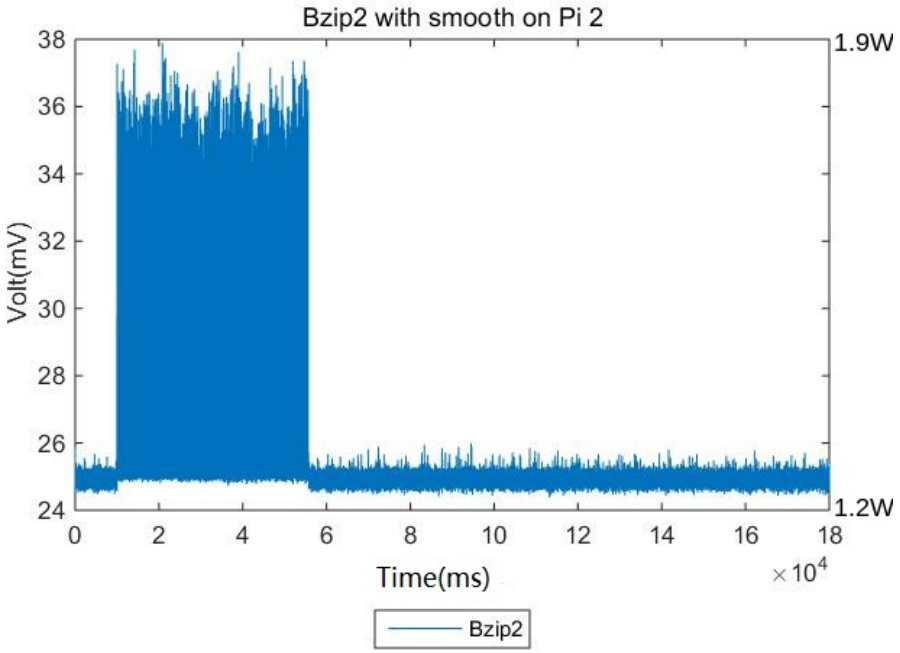

Figure 4.8: Bzip2 benchmark results on Pi 2

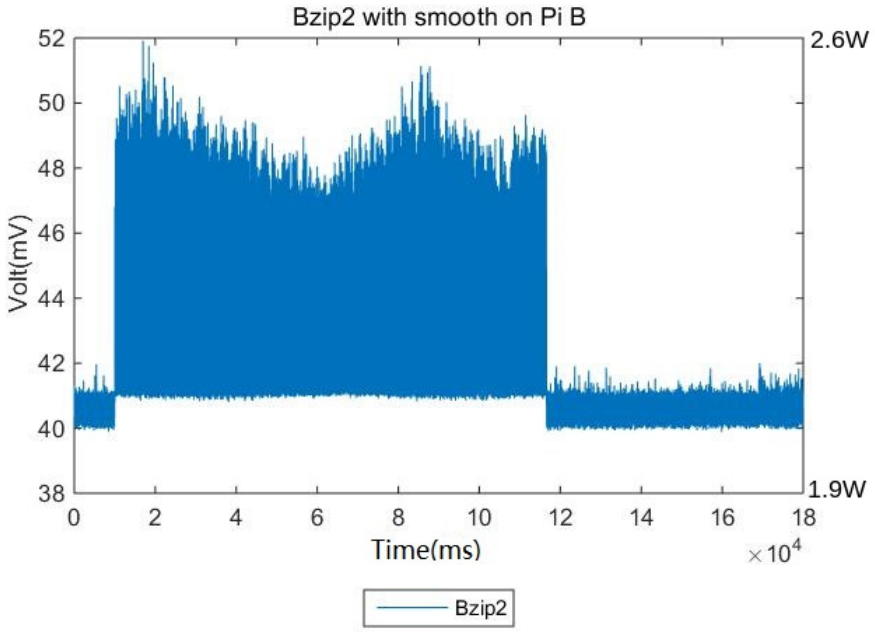

Figure 4.9: Bzip2 benchmark on Pi B

```
Performance counter stats for 'bzip2 -k -f ./input.source':

   19,226,843,140      instructions:u

     45.719224756 seconds time elapsed
```

Figure 4.10: Bzip2 perf results on Pi 2

```
Performance counter stats for 'bzip2 -k -f ./input.source':

  <not supported>      instructions:u

    106.644694545 seconds time elapsed
```

Figure 4.11: Bzip2 perf results on Pi B

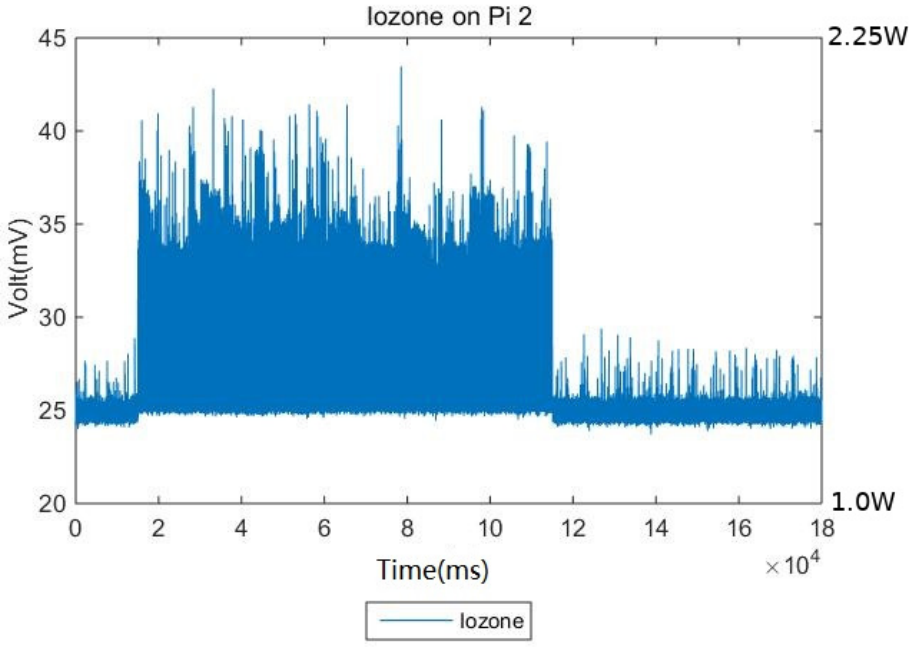

Figure 4.12: Iozone results Pi 2

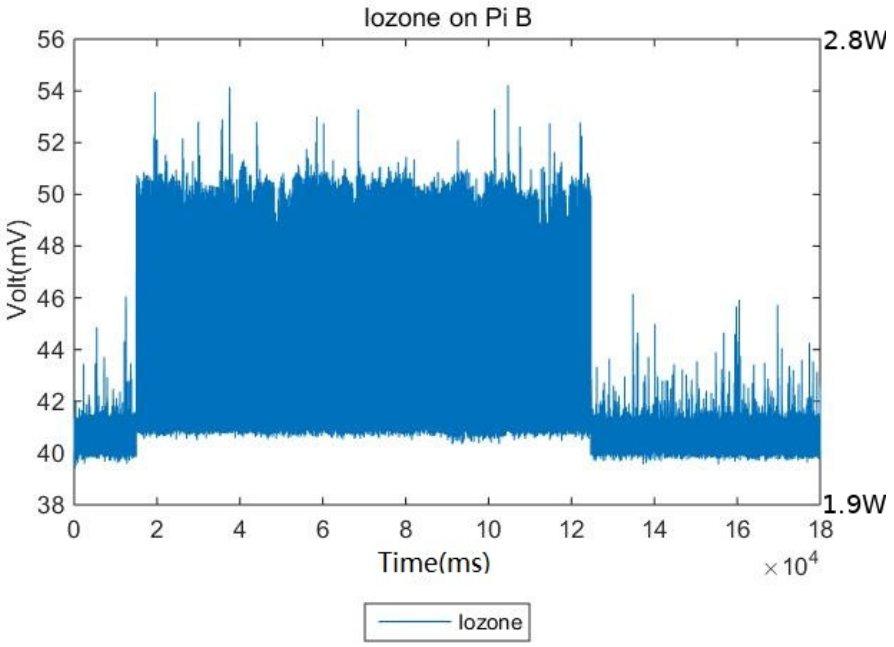

Figure 4.13: Iozone results on Pi B

```
Performance counter stats for './iozone -a -g 16M -i 0 -i 1':

      9607.831906      task-clock (msec)         #    0.096 CPUs utilized
            5,161      context-switches          #    0.537 K/sec
               14      cpu-migrations            #    0.001 K/sec
           12,395      page-faults               #    0.001 M/sec
    6,975,239,967      cycles                    #    0.726 GHz
  <not supported>      stalled-cycles-frontend
  <not supported>      stalled-cycles-backend
    2,667,325,330      instructions              #    0.38  insns per cycle
      294,431,835      branches                  #   30.645 M/sec
       41,564,151      branch-misses             #   14.12% of all branches

    100.096427025 seconds time elapsed
```

Figure 4.14: Iozone perf results on Pi 2

```
Performance counter stats for './iozone -a -g 16M -i 0 -i 1':

     17953.886000      task-clock (msec)         #    0.164 CPUs utilized
            3,506      context-switches          #    0.195 K/sec
                0      cpu-migrations            #    0.000 K/sec
           12,393      page-faults               #    0.690 K/sec
   12,470,880,124      cycles                    #    0.695 GHz                      (50.37%)
    3,924,794,969      stalled-cycles-frontend   #   31.47% frontend cycles idle     (50.18%)
    2,478,429,337      stalled-cycles-backend    #   19.87% backend  cycles idle     (49.94%)
    2,437,021,825      instructions              #    0.20  insns per cycle
                                                 #    1.61  stalled cycles per insn  (33.16%)
      353,537,624      branches                  #   19.691 M/sec                    (33.12%)
       39,455,033      branch-misses             #   11.16% of all branches          (33.75%)

    109.753450870 seconds time elapsed
```

Figure 4.15: Iozone perf results on Pi B

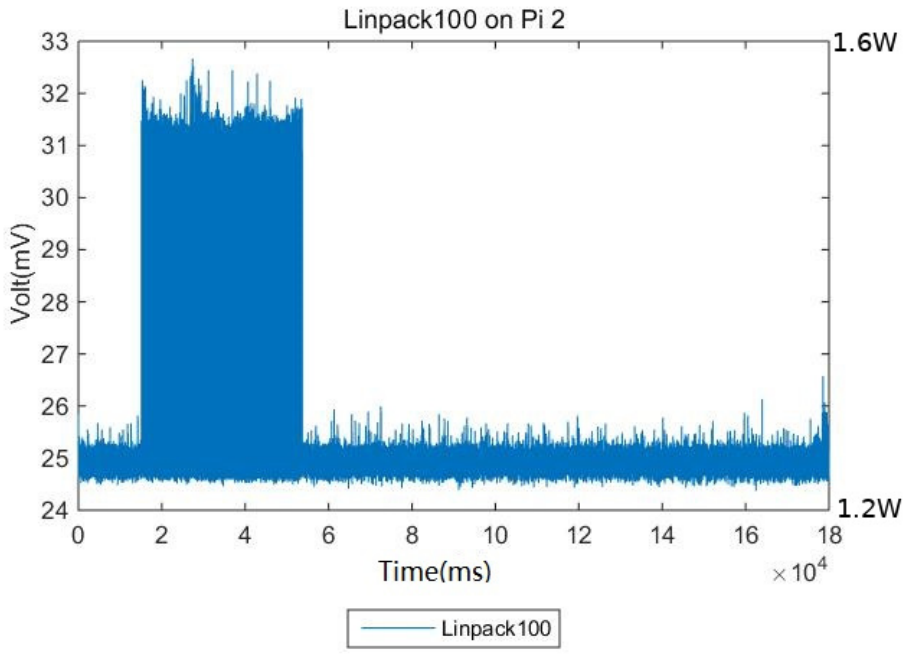

Figure 4.16: Linpack100 results on Pi 2

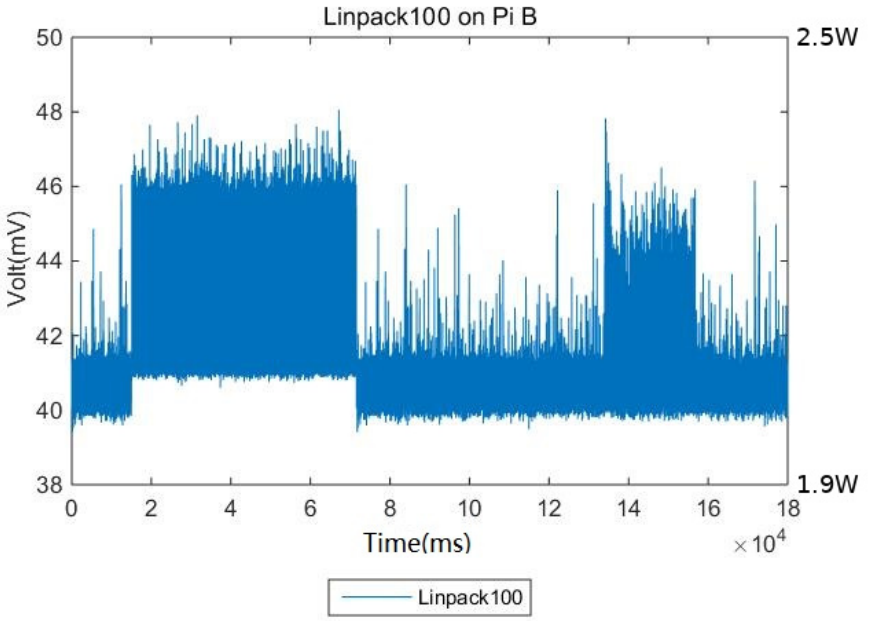

Figure 4.17: Linpack100 results on Pi B

```
Performance counter stats for './linpack':

     33848.485361      task-clock (msec)         #    0.874 CPUs utilized
               64      context-switches          #    0.002 K/sec
                0      cpu-migrations            #    0.000 K/sec
              142      page-faults               #    0.004 K/sec
   30,418,761,639      cycles                    #    0.899 GHz
   <not supported>     stalled-cycles-frontend
   <not supported>     stalled-cycles-backend
   15,477,781,200      instructions              #    0.51  insns per cycle
      472,214,143      branches                  #   13.951 M/sec
        7,397,707      branch-misses             #    1.57% of all branches

     38.735693704 seconds time elapsed
```

Figure 4.18: Linpack100 perf results on Pi 2

```
Performance counter stats for './linpack':

     20102.077000      task-clock (msec)         #    0.355 CPUs utilized
              318      context-switches          #    0.016 K/sec
                0      cpu-migrations            #    0.000 K/sec
              141      page-faults               #    0.007 K/sec
   14,007,513,090      cycles                    #    0.697 GHz                      (49.98%)
      222,648,806      stalled-cycles-frontend   #    1.59% frontend cycles idle     (50.04%)
       27,152,436      stalled-cycles-backend    #    0.19% backend  cycles idle     (50.06%)
    3,853,522,565      instructions              #    0.28  insns per cycle
                                                 #    0.06  stalled cycles per insn  (33.37%)
      126,302,772      branches                  #    6.283 M/sec                    (33.35%)
        3,380,627      branch-misses             #    2.68% of all branches          (33.32%)

     56.603093311 seconds time elapsed
```

Figure 4.19: Linpack100 perf results on Pi B

Chapter 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

This experiment had various goals, most of which were achieved.

The first and primary goal was designing a low-cost power measurement board. The equipment for the entire experiment, including Teensy, amplifiers and resistors, cost no more than $30. The two most expensive parts were a Teensy 3.1 and INA122 amplifier that cost $19.80 and $6.32, respectively.

The second goal was stability and long-term stability of the measurement board. We achieved this, with the device still working properly after reading Raspberry Pi data in idle mode for up to 1 day at a sampling frequency of 1kHz.

The third goal was high sampling resolution. The 12-bit resolution of the ADC was sufficient for most situations, but it does limit measurement accuracy especially at low voltages. There also appears to be noise in in the output, and it is unclear whether this is due to actual changes in power or if it is inherent in our measurement setup. Although the noise can be eliminated by post-processing, this may affect the accuracy of our measurements.

An additional goal was the ability to upload the data efficiently to another machine. The Teensy is severely memory constrained, thus it must regularly upload data to another computer. The transfer of data over the serial port is slow relative to the sampling speed. This means that every time we upload data to a computer, we lose a portion of the data. Even though there is not a lot of data in this section, it is a blow to the timeliness, completeness, and overall accuracy of the data.

Another goal was high sampling speeds. We manage 1kHz, which is many orders of magnitude faster than the 1Hz provided by some devices. However, the speed of

modern computer CPUs is in the GHz range, so when we test these computers we potentially lose a lot of detail. Efforts were made to raise the sampling frequency, but the cost of increasing the sampling frequency was a dramatic drop in stability. This is mostly a problem with our data-capture program, but there may be limitations inherent in the Teensy architecture as well.

A final goal was ease of use. After downloading the program to the Teensy, the user only needs to install the serial port driver and a serial port testing program to use it easily. Drivers can be downloaded from the official website of Teensy and the serial port testing program can be easily found on Google. The user interface text is human readable. The more difficult-to-understand parts have clarifying notes and examples available so that users can easily use it on the first try.

5.2 Future Work

There are several future items potentially worth doing.

The Teensy 3.1 used in this experiment has been discontinued, and the new Teensy models provide performance improvements. It would be good to try out these new models of Teensy.

There is a lack of trustworthy comparative data available for this experiment. We would like to test a number of additional competing devices that can provide similar power measurements, and analyze the differences in results gathered from these devices.

REFERENCES

[1] Bangor, ME electricity statistics. http://www.electricitylocal.com/states/maine/bangor/, 2017.

[2] Top 500 supercomputing sites. http://www.top500.org/lists/2017/06/, 2017.

[3] bzip2. http://bzip.org/ , 2018.

[4] iozone filesystem benchmark. http://iozone.org/ , 2018.

[5] IViny compact data acquisition device. https://github.com/ivmech/iviny , 2018.

[6] Teensy and Teensy++ pinouts for C language and Arduino software. https://www.pjrc.com/teensy/pinout.html, 2018.

[7] Units of energy. https://en.wikipedia.org/wiki/Units_of_energy, 2018.

[8] D. Bedard, R. Fowler, M. Linn, and A. Porterfield. PowerMon 2: Fine-grained, integrated power measurement. Technical Report TR-09-04, Renaissance Computing Institute, 2009.

[9] W. Bircher and L. John. Complete system power estimation: A trickle-down approach based on performance events. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 158–168, Apr. 2007.

[10] M. Cloutier, C. Paradis, and V. Weaver. A raspberry pi cluster instrumented for fine-grained power measurement. *Electronics*, 5(4):61, 2016.

[11] H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *ACM/IEEE International Symposium on Low-Power Electronics and Design*, pages 189–194, Aug. 2010.

[12] S. Desrochers, C. Paradis, and V. Weaver. A validation of DRAM RAPL power measurements. In *The International Symposium on Memory Systems*, Oct. 2016.

[13] A. Eames. RasPi power usage measurements ALL Models. `http://www.wattsupmeters.com/`, May 2012.

[14] A. Eames. Raspberry Pi2 – power and performance measurement. `http://raspi.tv/2015/raspberry-pi2-power-and-performance-measurement`, Feb. 2015.

[15] Electronic Educational Devices. Watts Up PRO. `http://www.wattsupmeters.com/`, May 2009.

[16] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron. PowerPack: Energy profiling and analysis of high-performance systems and applications. 21(6), May 2010.

[17] J. Geerling. Raspberry Pi power consumption. https://www.pidramble.com/wiki/benchmarks/power-consumption, 2009.

[18] T. Gleixner and I. Molnar. Performance counters for Linux, 2009.

[19] D. Hackenberg, T. Ilsche, R. Schoene, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2013.

[20] C. Moreno, S. Fischmeister, and M. Hasan. Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis. In *Proc. 14th ACM SIGPLAN conference on Languages, Comilers, and Tools for Embedded Systems*, June 2013.

[21] C. Paradis. Detailed low-cost energy and power monitoring of computing systems. Master's thesis, University of Maine, Aug. 2015.

[22] A. Petitet, R. Whaley, J. Dongarra, A. Cleary, and P. Luszczek. HPL — a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Innovative Computing Laboratory, Computer Science Department, University of Tennessee, v2.2, `http://www.netlib.org/benchmark/hpl/`, Dec. 2017.

[23] M. Rashti, G. Sabin, D. Vansickle, and B. Norris. WattProf: A flexible platform for fine-grained HPC power profiling. In *IEEE International Conference on Cluster Computing*, Sept. 2015.

[24] E. Rotem, A. Naveh, D. Rajwan, A. Anathakrishnan, and E. Weissmann. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[25] B. Schaff and V. Weaver. Sensing power consumption of desktop computer system components. Technical report, University of Maine, Aug. 2016.

[26] K. Singh, M. Bhadauria, and S. McKee. Real time power estimation and thread scheduling via performance counters. *Computer Architecture News*, 37(2):46–35, July 2009.

[27] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. http://www.specbench.org/osg/cpu2006/, 2006.

BIOGRAPHY OF THE AUTHOR

Yanxiang Mao was born in Beijing, China.

Yanxiang Mao is a candidate for the Master of Science degree in Computer Engineering from The University of Maine in May 2018.