A low cost power measurement device to improve energy efficiency of HPC devices

Project Report

Manuela Beckert Janosch Hirsch

August 5, 2015

Contents

1. Introduction

Energy efficiency means using less energy to provide the same service[1], like lighting a room with light bulbs that consume less energy. Or writing a program in a way that it finishes faster and consumes fewer resources. While energy efficiency plays an increasing role in our everyday life, for example how much money we could save each month if we would buy a new freezer with less energy consumption, it has the greatest importance in places where a lot of machines have to be powered all over the year, like data centers. According to an evaluation of the Borderstep Institut[2] from May 2012, the energy consumption of servers and data centers in Germany in the year 2011 was 9.7 TWh. That makes around 1.2 billion Euro of energy costs and would take 4 medium sized coal power stations to provide it.

Most energy in data centers and enterprise networks is consumed by servers and cooling facilities. Making them more efficient has the highest impact on costs and is therefore the main research subject. But the amount of energy consumed by networking devices will continue to grow, too, especially with the growth of infrastructures to support cloud computing and software as a service models[3].

In order to improve energy efficiency, it is necessary to measure how much power a machine consumes and where (most of) that power is consumed. If we know which component consumes the most power, we can replace it by a more efficient one and if we know which section of the software that might be running on the machine consumes the most energy, we can try to improve that section. Therefore, we need power measurement devices.

1.1. Motivation

Our objective is to measure the power consumption of code running on a computer. To achieve this, we need a measurement device which can measure power precisely and time dependent, so we can correlate the data to sections of the program.

We want to distinguish the power internally to know which parts of the computer are actually demanding the power. If we study the power cables of the main board, hard disk drives and graphics card for example, we can make a good guess where the power is needed.

The device shall not only be usable on a single computer, but in a whole cluster. So it has to be cheap, easy to set up (e.g. USB communication) and simple to use. It is not feasible if a watt meter costs 1000 Euro when we want to measure a cluster with 32 nodes – we would have to pay 32 thousand Euro.

Most commercial devices are not only expensive, but also hard to integrate into existing hardware or software. They are designed for laboratory conditions and not intended for a permanent integration. Even if they can be integrated, they are often so big that we cannot possibly put them into a server machine and close the machine afterwards. If we were to measure a whole cluster, they would take up a lot of additional space. Some devices are hard to integrate into the software, too. For example they need to be operated via an intermediate software, which only runs on Windows. Often such software is not open source so we can't modify it to fit our needs.

To overcome those limitations we developed a microcontroller based power measurement device that can observe, for example, mainboard, GPU and HDD power rails individually. It consists of an Arduino and a Hall effect sensor shield, which are cheap, small, easy to integrate and scalable. We integrated the device into the power measurement library pmlib.

1.2. Related Work

There are two main related works named PowerPack[5] and PowerMon2[6]. These as well aim to analyze the internal power lines in detail. PowerPack in fact monitors both the external and internal power lines to also be able to evaluate the power supply's efficiency. For inspecting the internal lines professional high precision measurement devices from Texas Instruments are used. These are super accurate and capable of delivering enormous sampling rates but extremely expensive and space consuming too. Thus infeasible to deploy in larger scales or outside the scientific or commercial scope.

PowerMon2 on the other hand is very similar to our project. Like we do, it focuses on monitoring internal power lines and uses a microcontroller based approach for this. The measurement process differs though as other sensing components are utilized. The main goals also are cost efficiency and scalability.

2. Hardware

The measurement device consists of an Arduino microcontroller board, a custom made measurement shield and connectors for the cables to be measured.

2.1. Arduino

An Arduino[7] is a microcontroller board of roughly the size of a credit card (Picture 1). It comes with preinstalled hardware and software which make it very easy to use even for beginners.

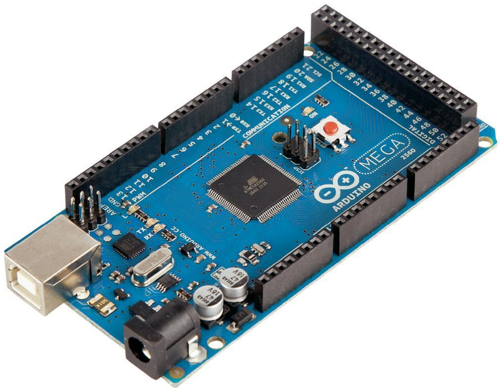

Figure 1: Photo of an Arduino Mega 2560[12]. The black square in the center of the the board is the microcontroller chip. The USB adapter is located at the very left. An external power source can be plugged in at the bottom. At the borders of the other three sides several general purpose pins are located.

A microcontroller chip is located in the center of the board. It will run our program. The Arduino has an on board bidirectional USB interface for communication with a computer, for example in order to load the program onto the chip. The board can be powered via USB or an external power adapter. There are several general purpose pins on the board which can be used as input or output, both analog and digital.

For our project we used an Arduino Mega 2560 which has 16 analog input pins, the highest count among the Arduino family. We will be using those 16 analog pins as input pins for the hall effect sensor output voltages. In consequence, our device will be able to measure 16 channels at a time. Since not all lines of a cable are actually interesting to measure (see chapter 5.1), those 16 channels are enough to measure the CPU, HDD and mainboard of a machine at the same time, so it will suffice for our purpose. Analog input signals need to be converted into digital values before the chip can work with them. For that purpose there is an analog to digital converter integrated in the microchip. The chip communicates via a serial interface with the computer and we will need to think about how to send values efficiently, as well as what protocol to use. This will be discussed more deeply in chapter 4.1.

Additional boards, called *shields* can be easily plugged on top of the Arduino, making it easy to combine with other controllers without any need to solder. An Arduino board provides many possibilities, but is still cheap (around 40 Euro), easy to use, open source, well documented, has a large community and is small enough to fit into a closed machine. Thus it perfectly satisfies all our requirements.

2.1.1. Arduino IDE

Arduinos provide their own simple IDE (figure 2). After selection of the Arduino model and desired serial port in the Tools menu, the code can be uploaded to the board by clicking just one button. The IDE is a little crude for a real programmer, as is does not offer any development tools like code completion or software libraries for advanced data structures, but it is very simple to use. There is a serial monitor to track data received from the Arduino. It interprets all data as ASCII symbols, thus not very useful if binary data is sent. The chip is programmed in a C like programming language. Again, this language is very simple and minimalistic to make it easy to use, but it serves our purpose. In figure 3 is an example for an Arduino program that makes the on-board LED blink.

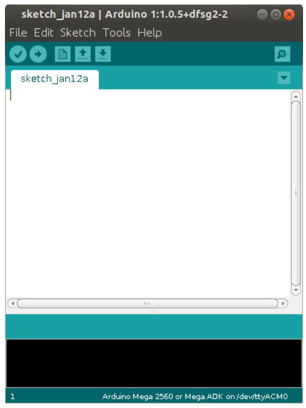

Figure 2: Screenshot of the Arduino IDE

Every pin is numbered and can be addressed by its number. To access a pin there are four functions: `analogRead, analogWrite, digitalRead and digitalWrite`, which read or write in analog or digital mode. While functions can be defined at will just like in C, there are two important special functions: `setup`, which gets called every time the Arduino powers up, and `loop`, which will be called after setup and, as the name suggests, just runs in a infinite loop, as long as the Arduino has power. There are more special functions for other events, like input data arriving at a serial interface. These act as interrupt functions in between the normal command flow.

```
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);    // turn the LED on
                              // (HIGH is the voltage level)
  delay(1000);                // wait for a second
  digitalWrite(led, LOW);     // turn the LED off by making
                              // the voltage LOW
  delay(1000);                // wait for a second
}
```

Figure 3: Arduino program for a blinking LED (an example program included in the IDE).

2.2. Analog-to-digital converter

An ADC (Analog-to-digital converter) converts an analog signal to a digital representation of it. Analog signals are continuous in the time and space domain. That means an analog signal can have any imaginable value at any imaginable point of time. This behavior can not be represented digitally. The digital domain is discrete, what implies that you can only have a finite amount of values and time intervals.

To get a digital representation of an analog signal it must be sampled and quantized.

Quantization is to transform coherent continuous values into discrete ones. To do so a finite and predetermined set of values is needed. In the quantization process every continuous value is set to the nearest finite one out of the set. The resulting error is called quantization error.

Sampling is the quantization of the time domain. The quantization of time continuous values is done in fixed time intervals. In every interval a quantization of the signal value is done. The number of quantized values per second is called sampling rate.

The number of bits an ADC can use for the quantization is called resolution and defines the range of discrete values. The sampling rate and the resolution determine the accuracy of the digital representation of the signal.

Figure 4 visualizes an analog signal and a resulting digital representation after sampling and quantization is done.

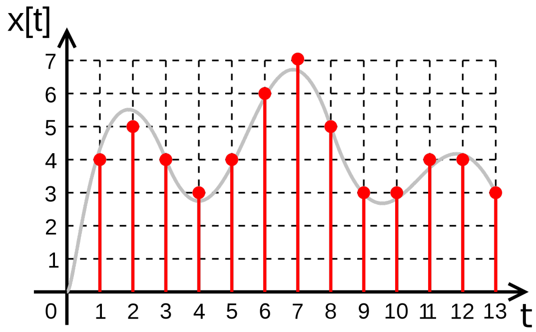

Figure 4: Sampling in the time domain and quantization in the space domain of the analog signal (grey line). The result is a digital representation of it (red dots).

2.2.1. Arduino ADC

The integrated ADC of the Arduino Mega 2560 has a resolution of 10 bits and can be feed trough 16 multiplexed analog input pins. The maximum sampling rate is roughly specified with 10,000 samples per second. However the Arduino ADC has to be called through a function for each sample. It does not run in a individual loop providing values to read out of a buffer. So the sampling rate is determined by the number of function calls per second and does not have to be periodic. Of course by having further computation between this function calls the sampling rate will decrease. Also multiplexed input channels imply that the sampling capability is shared by all used input channels.

2.2.2. Reference voltage

The Arduino ADC can quantize voltages from 0V to a maximum of 5V. This voltage range will be quantized into 10 bit values corresponding to 1024 levels with each level representing the voltage resolution of $\frac{5\text{V}}{1024} \approx 4.88\text{mV}$.

The ADC reference voltage can be used to shrink the voltage range but increase the voltage resolution. The reference voltage defines the maximum voltage to be quantized. The whole 10 bit resolution is used on the voltage range of 0V to the reference voltage. If the reference voltage is set to only 1.1V the voltage resolution would be $\frac{1.1\text{V}}{1024} \approx 1.07\text{mV}$. A higher voltage resolution leads directly to a higher ADC accuracy.

For an efficient use of an ADC it is important to use the whole working range. Otherwise the unused resolution is wasted. For the most efficient use the maximum voltage that should be convertible with the ADC should be estimated and used as the reference voltage. The lower voltage of the Arduino ADC voltage range can not be adjusted and is fixed at 0V.

For an accurate ADC quantization the reference voltage needs to be stable and known exactly, because it must be used to calculate the voltage of the digital values. The Arduino Mega 2560 provides two internal regulated voltages of 1.1V and 2.56V to choose. Additionally an ADC reference voltage input pin can be used to feed in a desired reference voltage. By default the 5V USB voltage is used unregulated. This just serves for rough quantizations. For the same reason you should only use the regulated 3.3V Arduino rail as source for a voltage divider resistor arrangement if you want to feed in a custom voltage.

The used reference voltage source option is defined in software.

2.3. Hall effect sensor

A Hall effect sensor can be used for measuring current through an electric circuit. To do so the Hall effect is utilized. The Hall effect describes a voltage that is occurring if electrons pass a non parallel oriented stationary magnetic field through a wire. This voltage is called Hall-voltage and has its polarity orthogonal to the electrons flow and the magnetic field. It comes off due to the Lorentz force appearing on an electron passing through a magnetic field. The Hall-voltage is proportional to the current through the wire and thus serves as value to calculate the current along with a constant composed of material constants and the magnetic field constant. Figure 5 illustrates the Hall effect principle.

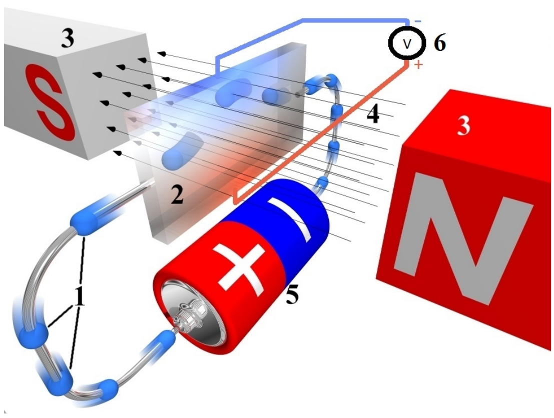

Figure 5: The Hall effect: 1 Electrons, 2 Hall-element, 3 Magnet, 4 Magnetic field, 5 Electric circuit, 6 Hall-voltage.

2.3.1. Hall effect sensor Integrated Circuits

Hall effect sensor ICs (Integrated Circuits) have an integrated Hall effect sensor and amplifying circuit. The Hall effect IC itself needs a supply of 3.3V to 5V to work. The electric circuit in which the current is to be measured is passed through the IC. On an output pin the very low hall-voltage is amplified to levels between 0V for no current and 5V for the maximum specified current the Hall effect IC is capable of.

2.3.2. Allegro ACS713

The measurement shield is equipped with 16 Allegro ACS713 Hall effect sensors. They were picked because of their appropriate technical specifications and especially because of their small size (surface mount package). The Allegro ACS713 can measure form 0 up to 20 ampere and is mainly used in the automotive area. However, on the output of this IC an offset voltage of about 0.5V is present if no current is passing through the Hall effect sensor. The ampere range of 0 to 20A is proportional to the output voltage range of 0.5V to 5V.

2.4. Arduino Shield

In figure 6 a photo of the assembled Arduino shield can be seen. It consists of 16 hall effect sensor ICs, some capacitors for signal smoothing and noise reduction and a connector for each channel.

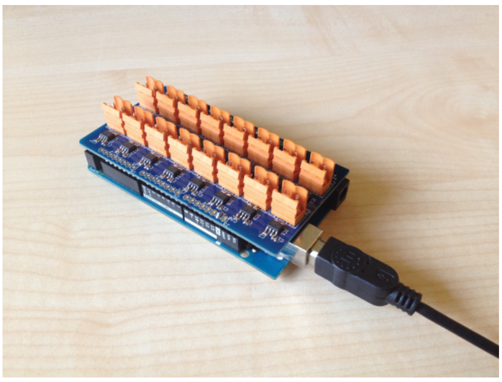

Figure 6: Photo of the Hall effect sensor shield plugged on top of an Arduino. The shield has 16 orange connectors for each channel soldered on top.

2.4.1. Wiring

In figure 7 the lines of all important PSU (Power Supply Unit) connectors in a server environment are displayed. The interesting lines to measure are the yellow, red and orange ones. Ground lines do not need to be measured because they carry the current of all other lines combined back to the power supply. All lines of different colors are mainly control lines and do not deliver power for the main processing units. The voltage of the lines is necessary to know for calculating the power. Yellow lines provide 12 Volts, red lines 5 Volts and orange lines 3.3 Volts.

Lines of the same connector and voltage can be merged together and thus measured through a single channel. This needs to be done if 16 channels are not enough to measure all desired lines. In our experiments all lines of the same connector and voltage had the exact same power characteristic so no information is lost by merging these. The ATX standard specifies multiple lines to distribute the over all power over multiple wires thus lowering the resistance and voltage drop. One has to be aware that merging lines increases the channels measured current what may effect the choice of the right ADC reference voltage described in the calibration section.

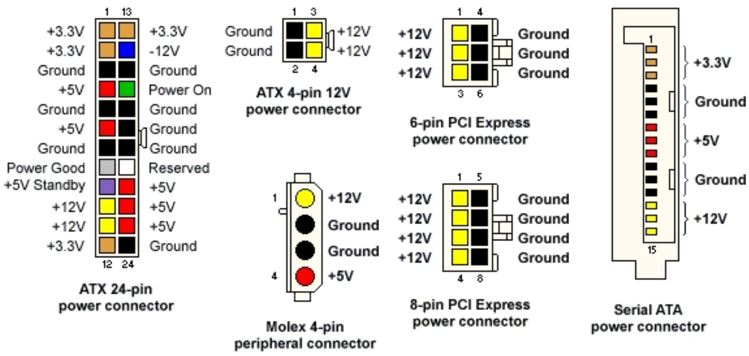

Figure 7: The lines of common PSU connectors.

Figures A.7.1 and A.7.2 show our measurement device installed into a desktop and server machine.

3. Calibration

The measurement system (Shield, Arduino and pmlib) as a whole needs to be calibrated to get exact current readings. The critical steps are the conversion of current to voltage by the Hall effect sensor ICs and the voltage quantization with a given reference voltage by the Arduino ADC. Both steps need calibration to be exact. We need to know which current corresponds to what voltage and what voltage corresponds to what quantization level. This can be calculated but will not correlate to the reality because of various reasons, e.g. manufacturing tolerances and environmental influences. To get the most accurate results in minimizing errors we calibrate both steps, which happen in series, as one. This means we relate currents directly to quantization levels.

The calibration process begins by determining a permanent reference voltage destined for the measurement scenario and is completed by calibrating currents thereafter. After calibration the system is in a fixed state and does not need to be calibrated again unless the measurement scenario changes to one in which a different maximum current value can occur. For example moving the device from a desktop to a server machine or upgrading the machine's hardware.

3.1. Determining the reference voltage

By measuring the currents of all single wires in a desktop and a server machine under full load we evaluated that in our scenario no wire is handling more than a maximum of 4 to 5 amperes. Thus the Hall effect sensor ICs will never exceed a corresponding output voltage to be quantized by the Arduino ADC. This maximum output voltage can be calculated by mapping the Hall effect IC's ampere range to its output voltage range as shown in figure 8. But to have the most accurate data, this voltage should be measured directly at the Hall effect IC's output pin with a voltmeter.

$$\frac{5V_{20A} - 0.5V_{\text{offset}}}{\frac{20A_{\text{IC max}}}{5A_{\text{scenario max}}}} = 1.125V$$

Figure 8: Calculating the Hall effect sensor IC's output voltage for a current of 5 amperes.

The optimal reference voltage for our scenario would be about 1.2V. Unfortunately this is a little higher and a lot less than the Arduino's internal provided voltages of 1.1V and 2.56V. For the most efficient ADC use and thus the highest resolution, we should have used an external reference voltage tapped of a voltage divider on the Arduino 3.3V rail. But we did not have enough time and no suitable resistors at hand, so we did the tests with the internal 2.56V as reference voltage. This roughly halves the resolution we could have. $\frac{1.2V}{2.56V} \approx 0.47$ Efficiency.

3.2. Channel calibration

All measurement channels consist of the same hardware, therefore the calibration of one channel should be enough to suppose the same characteristics for every channel. But because of manufacturing tolerances and other influences like solder joint quality we decided to do the calibration on each channel individually. This turned out to be a good decision as the regression lines of each channel differ slightly. The ADC should not have a great contribution in that because it is just one multiplexed unit, so the deviations must come from the measurement shield. Possible reasons could be the solder joint qualities and the manufacturing tolerances of the Hall effect sensor itself or its external passive components like capacitors.

3.2.1. Calibration setup

For the calibration process you need to be able to generate at least two different stable and known currents in an electric circuit. This can be done with either a variable voltage power supply and a dummy load or a fixed voltage power supply and different kinds of dummy loads. Most bench power supplies have a variable voltage. With the addition of a power resistor of for example 1 Ohm as load, sufficient current values inside the power specifications of the power supply and resistor can be generated. The current can be calculated with Ohm's law shown in figure 9.

$$I = \frac{U}{R}$$

Figure 9: Ohm's law: I Current, U Voltage, R Resistance

To get different currents with a fixed voltage power supply, different resistances are needed. A variable power resistor or multiple power resistors connected in series or parallel can be utilized. We used light bulbs as load and achieved various current values by adding bulbs connected in parallel. To read the current values an ampere meter is necessary.

All channels can be calibrated at once by connecting all terminals in series with the electric circuit. In that configuration every channel experiences the exact same current flow of the circuit. Figure 10 displays a wiring schematic for calibrating all channels at once.

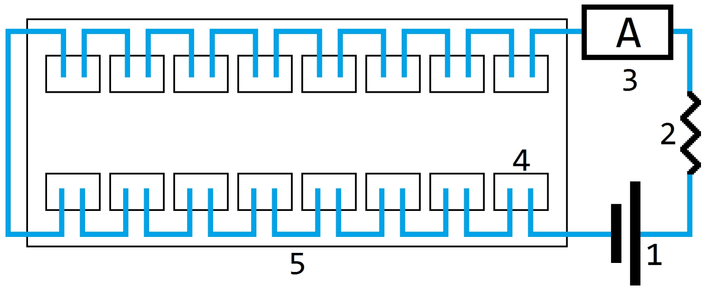

Figure 10: Measurement Arrangement: 1 Power supply, 2 Load, 3 Ampere meter, 4 Channel terminals, 5 Measurement shield

3.2.2. Calibration process

The channel calibration starts with a stable current in the electric circuit. Then the ampere value has to be read from the ampere meter and saved along with the quantization levels of all channels. This has to be done at least twice with different ampere values. With that data a linear regression of two points for each channel can be done. With more measured points the regression line will be more accurate but the data should be linear anyway because the Hall effect sensor ICs and the ADC specify linearity.

The slope and offset of the individual regression lines for each channel are necessary to calculate ampere values from quantization levels. These parameters need to be calculated from the data and saved in the pmlib settings file to complete the calibration. Figure 11 shows a plot of the regression lines we got through our calibration.

3.2.3. Calibration script

To make this process faster and easier we implemented a calibration script which automatically saves the quantization values for as many ampere values to be measured. It also samples a desired amount of times per current value and uses the mean value to eliminate noise. The linear regression for every channel is done automatically and the output can easily be copied and pasted directly into the pmlib settings file. Additional output data like the standard deviation can be analyzed to detect noise.

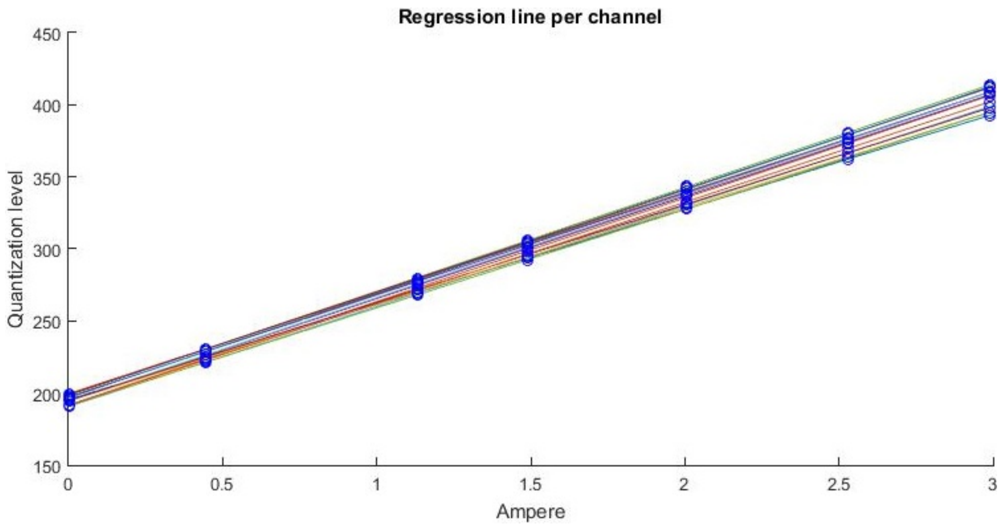

Figure 11: Regression lines

A guide of how to use the script can be found in appendix A.5.

4. Software

Figure 12 shows the setup for our measurement device. The device consists of the Arduino and shield. It measures the current on a client machine and offers adapters for HDD, CPU and mainboard for quick installation. The adapters are attached though connectors and can easily be replaced by other adapters if needed. The device reports the measured data to a server. On the Arduino runs a program which collects values from the shield and puts them into an efficient format to be sent to the server. We will explain the protocol we used, as well as the main idea of the program that runs on the Arduino. To evaluate the data received from the Arduino, we used the power measurement library pmlib. We will therefore also introduce the library and explain how we integrated the device.

4.1. Arduino: Protocol and Implementation

For communication of the Arduino with the computer, we need to send data over a serial interface, which transfers data bytewise. We want to maximize the sampling rate and the serial connection is one of the main two bottlenecks, so we want to minimize the amount of data we send. In consequence, we will have to tinker with

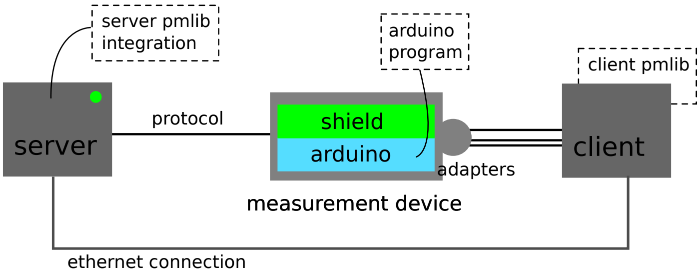

Figure 12: Distribution of hardware and software components for a measurement. Client and server can be the same machine.

bytes and bitmasks. The second bottleneck is the Arduino itself. Computation on a microcontroller will obviously take longer than on a regular machine. A single operation or two are still faster than sending a byte over the serial interface, but if we design the computation of the bytes we intend to send too complicated (e.g. with lots of integer shifts), the accumulated time can quickly surpass the sending time and thus yield a negative impact on the sampling rate. So we need some sort of protocol that is both easy (in terms of as few operations as possible) to implement on the Arduino side and at the same time compresses our data as much as possible.

There are two types of communication between the Arduino and the computer. First, the Arduino has to know which channels to measure. For that purpose, the computer has to transfer the channel list to the Arduino. This only happens once at the start of a measurement and is rather easily done with a bitmask. For 16 channels we just need to send two bytes, where every bit represents one of the channels. The bitmask

```
00000000 00001011
```

for example will mean that channels 0, 1 and 3 are to be measured. Extracting channels from the bitmask is quickly done on the Arduino side, too and since it only has to be done once, it will have no impact on the sampling rate even if we waste one or two operations here.

The second, and more interesting, type of communication is sending values from the Arduino to the computer. The channels have 10 bit resolution, so we need to send 10 bit for each value. We can only send bytes over the serial interface and sending two bytes for every value would waste 6 bit each time. It may not sound like a lot, but let's view it this way: 6 bit of 16 are about one third. If we used those 6 bits more efficiently, we could reduce the time needed to send the values by one third. Or the other way around: we could send 1.5 as many values in the same time. We tried to use those bits more efficiently thus.

If we do not concern ourselves with robustness, we could just use the leftover bits to start coding in the next value, like

```
AAAAAAAA AABBBBBB BBBBCCCC ...
```

where `A` is the first value, `B` the second one and so on. This is rather difficult to compute for different amounts of channels though, both on the Arduino and pmlib side, so we went with a slightly different approach where we put the highest two bits of every four values into an additional byte:

```
AAAAAAAA BBBBBBBB CCCCCCCC DDDDDDDD AABBCCDD ...
```

It still takes some effort to make it work for different amounts of measured channels, but is overall easier to implement. We tried out this protocol out of curiosity and on first glance it works very well. The sampling rate is maximized (7774 samples per second if all 16 channels are used) and in an experiment where we let the server run for two hours just collecting data, we always got correct values. The problems started when we tried to let the server actually do something with these values. Every time the client requested to receive the data that was collected, the bytes the server received started to be out of synchronization. With the bytes out of order, the bits were falsely shifted. This made everything we would receive afterwards total garbage. We assume the cause to be that the serial output buffer on the Arduino side overflows because the server misses to receive the values in time. Theoretically this should not happen because the program part that collects the bytes and the program part that sends the received data to the client are supposed to run in different threads. But it did happen. It might have been possible to fix this problem in pmlib, but our understanding of the library was not deep enough to do so. Even if we did find a fix, there was no telling what else might go wrong next. In the end, we didn't want an hour-long measurement to go to waste, just because someone stepped on a cable midway and a byte got lost.

We decided to go with a more robust protocol. We want to be able to tell whether a synchronization failure happened and to correct it. In order to tell which byte we are currently reading, we need to reserve at least one bit per byte for synchronization. We chose the first bit, set it to 1 if it is the first byte of a set of values, to 0 otherwise and used the remaining bits for the values like this:

```
1AAAAAAA 0BBBBBBB 00AAABBB
0CCCCCCC 0DDDDDDD 00CCCDDD
...
```

Note that there is a wasted bit in every third byte. It is theoretically possible to use that bit to code in another value. But we were not able to find any means of computation on the Arduino side that was not so complicated that it took longer than just sending an additional byte over the serial interface though. As soon as we used 32-bit integer shifts (instead of byte shifts), the runtime exploded. If we did not, we ended up with very long conditions inside a loop, and the runtime exploded. So we went with the protocol above at the end. This might be a point where the sampling rate could possibly be further improved.

We implemented both versions (with and without synchronization). Except for the `loop` method they are identical. There are only three methods:

- `setup`, which gets called when the Arduino is powered on and mainly just initializes the serial interface.

- `serialEvent`, which gets called every time there is incoming data on the serial interface. The only incoming data is the channel bitmask. This method extracts the channels from it and stores them.

- `loop`, which reads a value from each analog pin that shall be measured, codes them into bytes according to the protocol and sends those bytes via the serial interface.

We attached the `loop` functions of both protocols for reference in appendix A.1 and A.2.

For communication via a serial interface you need to choose a baud rate. The baud rate specifies the number of symbols transmitted per second. The serial interface uses bits as smallest unit so the baud rate specifies bits per second. The baud rate has to be equal or higher than the data-rate our protocol is capable of. We chose 115,200 baud equaling to 14,400 bytes per second. The synchronization protocol needs at worst 2 bytes to send 1 sample. But only if just one channel is measured. The serial interface's data-rate would only be able to deliver $\frac{14{,}400}{2} = 7200$ samples per second. However if more than 1 channel is measured the protocol efficiency is never worse than 1.7 bytes needed per sample. This corresponds to a maximum possible rate of $\frac{14{,}400}{1.7} \approx 8470$ samples per second through the serial connection, which is more than the Arduino is capable of sending.

With the synchronized protocol we achieve a sampling rate of 5879-7652 samples per second, depending on the number of channels we measure (figure 13).

Note that the y-axis goes from 5800 to 7800, the values are all rather similar thus. The sampling rate with just one channel is low as expected, as we need to send two bytes for every single value. With an even number of channels our protocol is most efficient hence we can note peaks for even channel numbers. The more channels we measure,

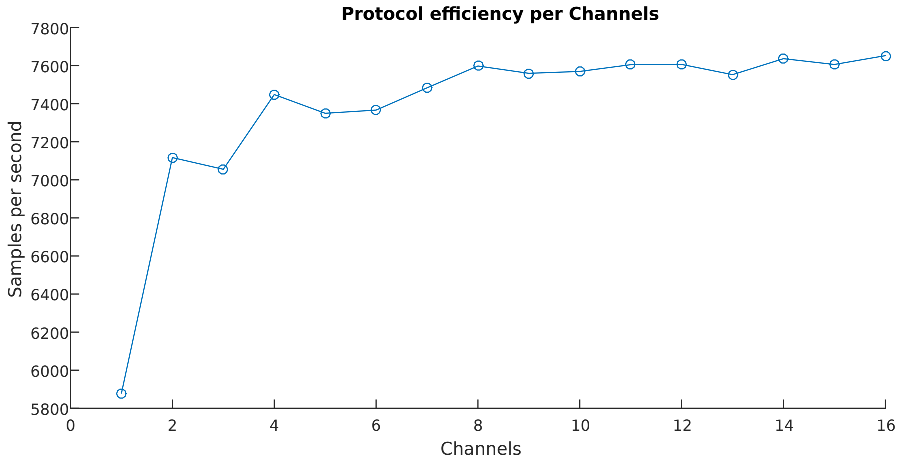

Figure 13: Samples per second in total with the synchronized protocol for different numbers of measured channels.

the more insignificant an additional byte that we send becomes, so the difference is not as noticeable for higher channel counts. In addition to the different efficiency for an even and uneven amount of channels, there is also an increase in efficiency with increasing channel count. This surprised us at first, because theoretically the steps we do for packing two channels should be the same, regardless of whether these two channels are part of a set of two or sixteen.

We assume it has to do with our use of loops. In our implementation (appendix A.1) we use two for-loops to iterate over the number of channels: first to retrieve time-bound values from the analog pins and thereafter to write the data to the serial interface. The body of the loops should have no effect on our sampling rate, as it does the same whether we have two or sixteen channels to measure. What differs though, is the amount of times we have to check the break condition. With only one or two channels to measure, the last condition check to break out of the loop has a greater over all computation time contribution than if we were measuring sixteen channels:

```
16  x measuring  1 channel   =  16 times * 2 loops *  2 checks  =  64 checks
 8  x measuring  2 channels  =   8 times * 2 loops *  3 checks  =  48 checks
 4  x measuring  4 channels  =   4 times * 2 loops *  5 checks  =  40 checks
 2  x measuring  8 channels  =   2 times * 2 loops *  9 checks  =  36 checks
 1  x measuring 16 channels  =   1 time  * 2 loops * 17 checks  =  34 checks
```

So for measuring one channel we need twice as many checks for the same amount of values as with sixteen channels.

As expected the sampling rate of every channel decreases with an increased amount of channels to measure (figure 14). There is not an exact increase of a factor of 16 for measuring just one channel instead of sixteen. This is because our protocol is more efficient for a higher amount of channels to measure. But the correlation of sampling rate per channel and number of channels measured is still clearly visible.

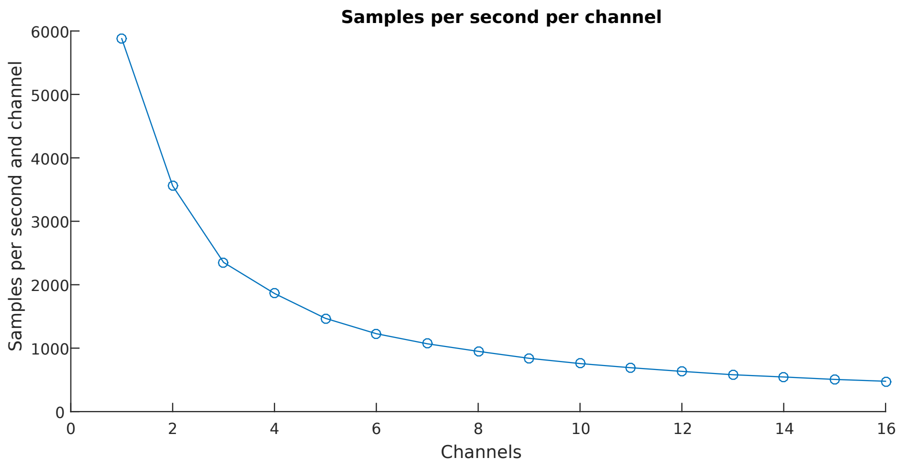

Figure 14: Samples per second *per channel* with the synchronized protocol for different numbers of measured channels.

We compared the efficiency of the two protocols in figure 15. The exact values can be looked up in appendix A.3. We can see that the first protocol (orange) is most efficient for multiples of four channels to measure, what makes sense because then the amount of bits we waste in order to deliver the highest bits is minimized. It is also notable that for one channel, both protocols are more or less identical, and for two channels the synchronized protocol (blue) is better, likely due to the fact that we used `byte` instead of `integer` at some points and achieved a small speedup with that. In all other cases the the synchronized protocol is about 100 samples per second slower, because of lower byte per sample efficiency. If we consider the scale though, that is a small price to pay for synchronization in our opinion.

4.2. pmlib

Pmlib[11] is a power measurement library developed by the work group *Wissenschaftliches Rechnen* (Scientific Computing) at University of Hamburg. It provides a framework for retrieving data from power measurement devices and processing it into a parseable or human readable form. Furthermore it provides methods which can be called right out of the source code of a program, which enables the programmer to

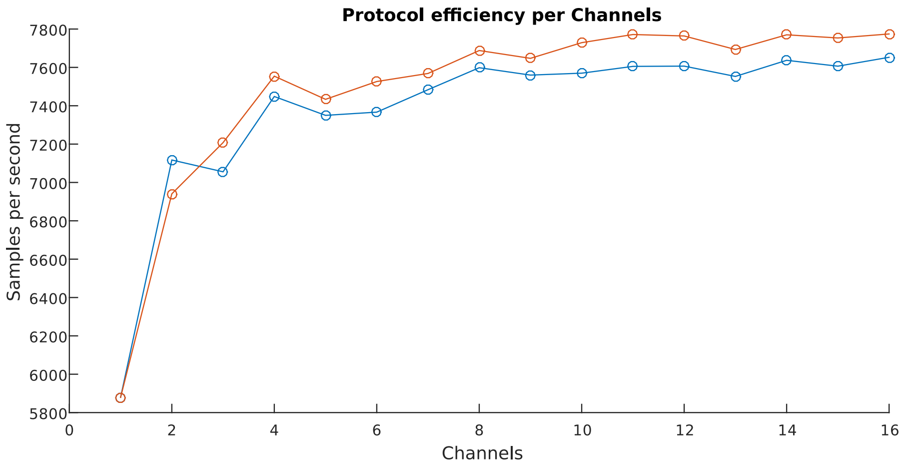

Figure 15: Comparison of samples per second for both protocols with different numbers of measured channels. Blue: synchronized protocol. Orange: unsynchronized protocol.

measure specific parts of his code. The library consists of two parts: server and client (figure 16).

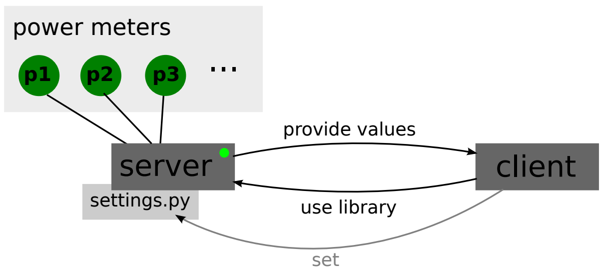

Figure 16: Overview of pmlib.

The server daemon has access to power measurement devices, collects data from them and sends the processed data to clients. A client library provides communication with the server and allows clients to measure the power consumption of their programs.

On the server side, there exists one class for every device that is, or can be, attached. That class handles how information from the device shall be interpreted, for example computing the power from the read data or storing them in a matrix. The idea is that a client has the power measurement device attached and if he asks the server for measuring, the server will start collecting data from the desired device. The information what device to read from and which channels on that device should be read is stored in a settings file named `settings.py` on the server side. The server reads the settings file on startup, begins collecting data when the client tells him to, process it according to the device's class and stops when the client tells him to. The collected data is then held until the client asks to receive it.

The client will have to set the needed information in the `settings.py` and can then just use the library methods in some program source code to start measuring. The server is written in Python, but there is a client library for C. If a client wants to retrieve data from a device "DCM1" for example, he only needs to call 4 methods, what, simplified, looks like this:

```
pm_create_counter("DCM1",...)
pm_start_counter(...)
[Code to measure]
pm_stop_counter(...)
pm_get_counter_data(...)
```

Pmlib also offers a whole collection of additional functions to the client, for example to determine which devices and channels are available to read from, selecting channels of interest[1] or receiving further information about the device. So the measurement can be as simple or as customized as the client wishes.

4.2.1. Integration of our device

An excerpt of the pmlib structure with all files that we added or changed is shown in figure 17. Firstly, we of course had to add a class for our device, `ArduPowerDevice.py`. This is the main file and contains all the logic needed for the protocol, retrieving and processing data and so on. It implements the interface `Device.py` from the same folder. In addition, there were two other files on the server side we had to touch. `__init__.py`, where we have to introduce our new device to pmlib, which is just one line to add:

```
from ArduPowerDevice import *
```

And, of course, the `settings.py`, where we need to tell pmlib that it shall retrieve data from our device and what channels shall be read. Again,

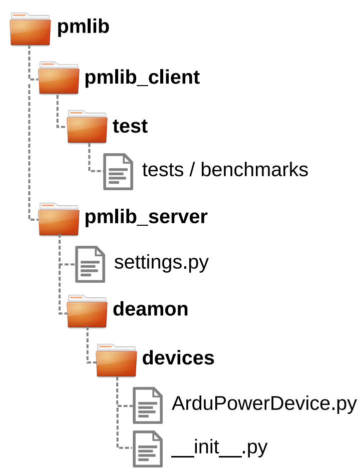

Figure 17: Relevant part of pmlib for our project.

[1] Of the channels that are specified in the `settings.py`.

there are just a few lines we needed to add. First,
to register our device

```
c=  ArduPowerDevice(name="ArduPowerDevice", computer=localhost,
    url="/dev/ttyACM0", max_frequency=100)
```

whereas the url has to match the USB port we attached our device to. And secondly a line

```
c.add_line(number=0, name="", voltage=3.3, description="",
           slope=71.4943627331, offset=198.6262)
```

for each channel to be measured. Each channel has a number. For computing the channel power of the retrieved data, we need to provide the line voltage of the channel, as well as the values from calibration (see section 3). An example where all 16 channels are registered is attached in appendix A.4.

In the main file, `ArduPowerDevice.py`, we had to implement the Device interface, so there are two main methods: `__init__`, which just sets some starting values, and `read`, which had to be implemented in a way that it yields a set of values (one value for each channel) every time it is called. We did this by setting up the serial connection and then running an endless while-loop that reads and processes a set of values from the serial interface and yields it. The use of `yield` brings the desired effect of getting a set of values each iteration.
The remaining code is mainly communication with the Arduino, synchronization and the protocol. As already mentioned in section 4.1, the server needs to inform the Arduino of the channels it is supposed to measure. This happens once at the start, after opening the serial connection.
Synchronization checks happen after every read in the loop. If a synchronization fail is detected, the whole set gets discarded and the beginning of the next set is searched for. As soon as a proper set (first byte starts with 1 and all others with 0) is found, the loop continues normally. For debugging we also print out a message on the command line if a synchronization fail is detected and corrected.
While the protocol is fairly easy to implement on the Arduino side, it took a lot of effort to translate the bytes back into values on the pmlib side. We spent a lot of time on this, especially since we tried out a lot of protocols. In order to extract a value from a sequence of bytes we need to know which bytes carry the bits we are interested in and how to shift them in order to retrieve the bits. These positions and shifts are somewhat periodic but we could not break them down to a simple formula, especially for the first protocol. We went with a solution with two arrays in the end and sticked to it with the synchronized protocol, too. The arrays are descriptively named *positions* and *shifts*. We iterate over the count of values we intend to retrieve. The byte containing the lower 7 bits of a value can easily be found by using the loop counter. For the byte containing the 3 higher bits we look up its position in the positions array and use

the shifts array to extract the bits properly. We did not have time to revise our code afterwards or to think of a better solution, or at least improve the readability. So while it works, our pmlib code is not very pretty. We would recommend to revise it prior to any future work with it.

5. Experimental results

To test our device, we installed it into a desktop machine, and later a sever machine, performed several benchmarks and collected power measurement data each time.

5.1. Testing Machines

For the tests we used a standard desktop machine with an Intel i7 (Sandy Bridge) Processor and a Nvidia Quadro graphics card. The server was a NUMA (non-uniform memory access) machine with a Super Micro Computer mainboard and two Intel Xeon X5560 Processors. While performing our tests Hyper-Threading was disabled on both machines.

5.2. Benchmarks

We used different benchmarks to stress different components of the machine. An overview of all benchmarks we used can be found in table 1.

part	**benchmark**
CPU	Linpack
HDD	hdparm
memory	STREAM
network	iPerf
user application	partdiff-par

Table 1: List of used benchmarks.

Linpack[13] stresses the CPU by letting the system solve a dense system of linear equations `Ax = b` on double precision. Its main purpose is to approximate how fast a computer can solve problems, but it also gives a good estimation on peak performance. We used a version called xlinpack that was provided to us by the work group *Wissenschaftliches Rechnen (WR)* at University of Hamburg.

Hdparm is a command line utility to set and view hardware parameters of hard disk drives, but it can also be used as an HDD benchmark with the command:

```
hdparm -t /dev/sda
```

STREAM[14] is a benchmark that measures memory bandwidth (in MB/s) for simple vector kernels. It was specifically designed to work with data sets that are much larger than any available cache. There were several values which could be adjusted, but we just set `STREAM_ARRAY_SIZE` to 7 million as per instructions in the STREAM file and compiled it with -fopenmp to make it run parallel.

IPerf is a network testing tool that can create data streams (TCP, UDP) and measure the throughput of the network that is transferring them. It can be used to measure network performance. We used it with the following parameters:

```
iperf --client [ClientIP] -t 60
```

Lastly, partdiff-par is a program developed on TU Muenchen by Prof. Dr. Thomas Ludwig (et. al.), who is now part of the *WR* group at University of Hamburg. It is a partial differential equation solver used as a lecture program for parallel programming. We used it as exemplary HPC application program because it does a lot of computation on multiple cores and its computation time can be influenced in every possible detail with parameters. The program roughly repeats two phases in a loop: computation of the matrix and writing it to the disk.

In addition to the data our device reported during the benchmarks, we also collected sensor data of the machines with a special version of the pmlib server called pmlib_server_west.

We did not find a scientific GPU benchmark and did not run any extensive GPU tests thus, but we found a program called Heaven[16] which renders a 3D graphic.

5.3. Results

The sensor data we collected while running the benchmarks as well as most of the beforehand mentioned benchmarks provided no enriching or useful data. So we focus on the productive results in this section.

In studying the power measurement data we observed and verified some typical behavior of certain power lines. The most interesting benchmark results are the ones of the partdiff-par program. The synthetic benchmarks only stress a distinct component so that no real interaction of components or different runtime phases can be observed.

5.3.1. Power lines

The CPU activity can easily be determined by inspecting the 12V ATX line. If the power level rises the CPU is busy calculating. Whenever the calculation phase is paused or terminated the unused inactive cores enter in a power saving idle state, which decreases the power demand drastically. With all cores in idle state the desktop machine's CPU takes around 20 Watt and the server machine's two CPUs around 70 Watt. On both machines every single calculating core needs about 15 Watt additionally.

On the desktop machine the 12V ATX line is very strongly correlated with the 5V mainboard line. This means whenever the CPU is calculating the mainboard is working, too. That is not very surprising, but out of the three mainboard voltages it only uses the 5V line. The 12V and 3.3V lines remain constant over a whole program run. So most of the processing related components are powered by the 5V line. Like the CPU power increases with more cores active, so does the 5V mainboard line: approximately 3.5 Watt per active core and around 10 Watt total with all cores idling.

On the server machine the mainboard behaves in almost the same way but predominantly uses the 12V line. On the 3.3V line, hardly any correlation is noticeable and it is nearly constant at roughly 7 Watt. The power of the 5V line behaves different than all other lines. By comparing the graphs of figures A.6.4 and A.6.6 it seems like the power decreases with the number of active calculating cores, since in CPU idle phases the power is at a constant value of about 9 Watt. This may be explainable electrotechnically. However, even idling the desktop mainboard consumes more power of the 5V line while only supporting one CPU. So in contrast the server mainboard works mainly of the 12V line.

Because about 16 Watt of power consumed by the desktop mainboard via the 12V line is a considerable amount we tried to figure out what its purpose could be. We found out that the power is even further increasing and correlated with stressing the PCI express graphics card. This can be observed in figure A.6.11 where we captured the launch of the graphics benchmark program "Heaven". It would be interesting to measure the power of the 12V line with a detached graphics card to see how far the power drops and what portion other components consume.

We were not able to investigate the purpose of the 3.3V line by experimenting. Traditionally the 3.3V lines were for powering the CPU and Memory. But today these require much lower voltages and at least the CPU requires a lot more power, so higher voltage lines with on board regulators are utilized[17].

The hard disk is using the 12V and 5V lines. The 3.3V line is specified in the SATA power connector standard but not used by the majority of hard disks[18] like the one in our desktop machine. By inspecting the measurement results, very high power spikes on the 12V line of the hard disk can be seen. They may be caused every time the actuator is accelerating the arm on which the heads are mounted to park or move them. The constant power of the 12V line may be used by the motor for spinning the platters. The 5V line is for the logic circuitry and the heads. When the hard disk is working, the power through the 5V line rises while the 12V line remains almost constant after the arm was moved. After performing the IO operation the arm is parked and the logic goes into some kind of idle state where less power is needed.

5.3.2. Partdiff

The characteristic of the partdiff application is transfered perfectly to the corresponding power graphs. Every iteration of matrix computations is equivalent to a high on the 12V ATX CPU line. In between these, the IO phase in which the matrix is written to the hard disk is happening. This can be traced ideally on the hard disk power lines. While doing IO the CPU is not needed and its power consumption decreases radically. There may be some activity noise of operating system processes running in parallel especially on the desktop machine.

By increasing the number of cores provided to the program the matrix computation phase finishes faster because the work is distributed across the cores with little overhead that does not yet have a great impact on a stand-alone machine. But of course with more cores involved the power requirement rises because active cores are not idling in a power saving state anymore. None the less the most efficient CPU usage is always to load all cores so that no idling energy is wasted. Even with more power necessary for the task normally it finishes faster so that the over all energy consumption is lesser.

The IO phase always takes the same amount of time because the hard disk always needs the same time to write the matrix having constant dimensions. Because this is not done asynchronously the computation and IO phase can be observed well separated in the power graphs. At the beginning and the end of every IO phase the hard disk's 12V line has large spikes due to moving and parking the heads. Also while working, the 5V line is noticeably loaded. In between this sections the computation phase with high load on the CPU and mainboard is occurring. The hard disk of the server and desktop machines behave very similar but are not of the same model so the characteristic slightly differs.

5.3.3. Linpack

The data captured of the linpack benchmark on the desktop machine is valuable for verifying that the CPU is exclusively powered via the 12V ATX line and for comparing the maximal power extend with the partdiff application. In figure A.6.7 the linpack benchmark was run with all four cores on the desktop machine. The power of the 12V ATX line is quickly increasing to about 110 Watt and roughly staying there with some minor downward peaks. This behavior is not correlated with any other power line, especially not with the 5V mainboard line. Furthermore its average power drawing is lower than in the partdiff run shown in figure A.6.3. However, the power drawing of the CPU is way higher when running the linpack benchmark. This basically tells that the partdiff application is not using the CPU with highest efficiency. But in real applications this is rarely the case whereas synthetic benchmarks are purpose-built for that.

On the server machine the exact same properties can be seen if Linpack is ran on one core as seen in figure A.6.8 or up to 3 cores. However, starting with four cores like in figure A.6.9 valleys in the 12V ATX line, and thus in the CPU activity, are occurring at the end. With eight cores as shown in figure A.6.10 this gets the norm and constant ups and downs in the CPU activity are present. None the less it is drawing more power than the partdiff application with the same number of cores as shown in figure A.6.6. Again anti-proportional ups and downs are occurring on the 5V mainboard line like mentioned before so it is definitely no IO activity.

5.3.4. Energy consumption on Server and Desktop

Figure 18 shows the energy consumption of runs of the partdiff application on different machines with several numbers of cores. On the x-axis S stands for server and D for desktop. The digit indicates the number of cores provided. Above the application runtime in seconds is shown. What stands out first is that the desktop machine is more than double as energy efficient as the server machine. Secondly using two cores on the desktop machine seems to be more energy efficient and faster than using four cores. This is due to a falsified measurement. Unfortunately the operating system scheduled another job in parallel to the measurement run of four cores. This and the fact that four cores would be faster with less energy consumption than two cores can be clearly seen in a broader perspective of the power line graphs. Even though the run with three cores is also slower than with only two no obvious interferences can be seen in the data. We did not do a full evaluation of the data immediately after capturing it so to get the exact results the measurement would need to be repeated. In addition the run with 7 cores on the server machine steps a bit out of line in its energy consumption although the runtime seems quite reasonable.

The largest amount of energy is consumed by the processors. The seconds most energy is required by the mainboard. In particular via the 12V line especially on the server machine. On the desktop machine it is mainly used for the PCI express graphics card. The energy amount may be reduced quite a bit by detaching it. The hard drive is consuming the least amount of energy which could also be reduced by using solid state disks.

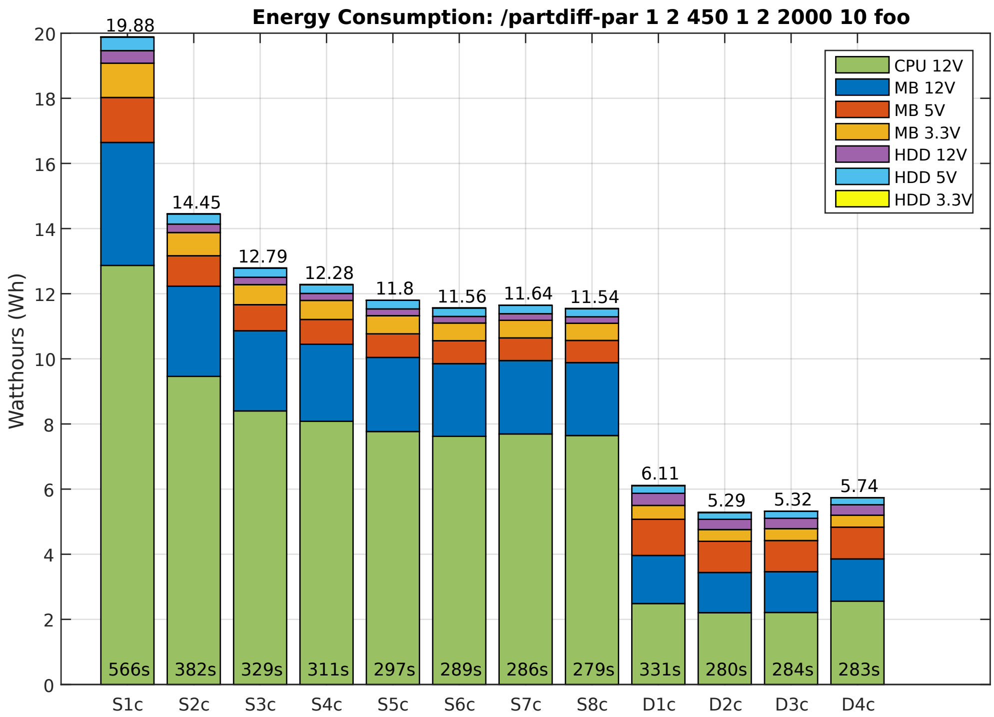

Figure 18: The energy consumption of a partdiff run with different numbers of cores on different machines. At the bottom the runtime is printed.

The reason for the desktop machine to be twice as energy efficient as the server has to do with the processor model. Nearly 50 percent of the energy is consumed by the processors. The server has two processors with 4 cores each, while the desktop machine has only one with 4 cores. By comparing the bars `S8c` with `D2c` it is apparent that the desktop processor must be at least twice as fast and almost twice as energy efficient than the server processor.

6. Conclusion

After working on the device and evaluating the measured data we are absolutely certain that the goal of a cheap, small and scalable power measurement device on microcontroller basis is practicable and achieved by this project. The results we got are very well suited to analyze energy efficiency of high performance computing machines. The achieved resolution and sampling rate is perfectly adequate for analyzing code running on a machine. We had no difficulties in recognizing code sections in the power graphs. Furthermore the interaction of different computing components and the characteristics the hard disk drive are precisely identifiable. Finally the integration into the power measurement library greatly eases the utilization of the device.

Of course can the device be further improved. A good improvement would be a resistor array which offers the user to choose from a variety of given reference voltages. The reference voltage does not need to be chosen frequently but a fine tuned one directly leads to a higher accuracy of the device and its measurements. Also can be thought about a case and easier to use connectors to simplify the fitting into a closed machine and speed up the setup process.

By optimizing the protocol on the Arduino side it should be possible to enhance the performance just a little more. But the main limitation is the microcontroller itself with its ADC resolution and computation capability. More advanced and precise parts would lead to higher costs so a trade-off has to be done.

All in all the resulted measurement device is quite practical and informative power readings to analyze energy efficiency can be done with ease.

A. Appendix

For easier comparison we used the same syntax highlighting as the Arduino IDE for the Arduino code.

A.1. Arduino Code Loop - Synchronized

```
void loop()
{
    // seperate loop to have the channel readings closer together timewise.
    for(byte i = 0; i < channels_to_read; i++)
    {
        values[i] = analogRead(read_channels[i]);
    }

    if(channels_to_read != 0)
    {
        // First Iteration
        Serial.write((values[0] | 0x80));

        high = (high << 3) | (values[0] >> 7);
        if((0 & 0x01) == 0x01 || 0 == channels_to_read_0)
        {
            Serial.write(high);
            high = 0x00;
        }
    }

    // Successive Iterations
    for(byte i = 1; i < channels_to_read; i++)
    {

        Serial.write((values[i] & 0x7F));

        high = (high << 3) | (values[i] >> 7);
        if((i & 0x01) == 0x01 || i == channels_to_read_0)
        {
            Serial.write(high);
            high = 0x00;
        }
    }
}
```

A.2. Arduino Code Loop - High Sampling Rate

```
void loop()
{
    // seperate loop to have the channel readings closer together timewise.
    for(int i = 0; i < channels_to_read; i++)
    {
        values[i] = analogRead(read_channels[i]);
    }

    for(int i = 0; i < channels_to_read; i++)
    {
        Serial.write(values[i]);
        high = (high << 2) | (values[i] >> 8);
        if((i & 0x03) == 0x03 || i == channels_to_read_0)
        {
            Serial.write(high);
            high = 0x00;
        }
    }
}
```

A.3. Comparison: samples per second

channels	synchronized	unsynchronized
1	5879.656594	5879.147156
2	3558.527375	3470.376750
3	2351.857441	2402.582930
4	1862.065905	1888.582845
5	1469.976028	1486.690182
6	1227.779860	1254.388576
7	1069.086516	1081.192215
8	949.786798	960.9935620
9	839.985461	849.693983
10	756.986302	772.894435
11	691.386038	706.494980
12	633.887713	646.994169
13	580.990068	591.796035
14	545.490571	554.996057
15	507.091368	516.896438
16	478.292542	485.896501

Table A.3.1: Comparison of samples per second per channel for the two protocols. They are plotted in figure A.3.2 below. For figure 15 we multiplied the values with the channel number.

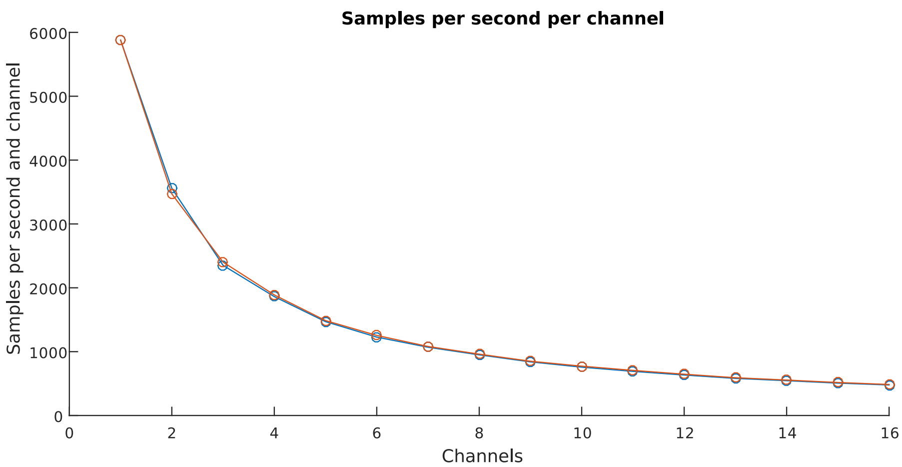

Figure A.3.2: Comparison of samples per second and channel for both protocols with different numbers of measured channels. Blue: synchronized protocol. Orange: unsynchronized protocol.

A.4. settings.py

Devices section of the settings.py with all 16 channels.

```
#=======================================================================
# PowerMeter daemon settings
#=======================================================================

# ...

#-----------------------------------------------------------------------
# Devices section
#-----------------------------------------------------------------------

ard= ArduPowerDevice(name="ArduPowerDevice", computer=intel5, url="/dev/ttyACM0", max_frequency=100)
ard.add_line(number= 0, name="24p-01c-3.3v",    voltage= 3.3, description="", slope=71.4943627331, offset=194.12)
ard.add_line(number= 1, name="HDD-SAT-5v",      voltage=   5, description="", slope=69.9332761719, offset=192.67)
ard.add_line(number= 2, name="24p-02c-3.3v",    voltage= 3.3, description="", slope=72.0016991302, offset=193.11)
ard.add_line(number= 3, name="HDD-SAT-12v",     voltage=  12, description="", slope=70.8100589449, offset=193.07)
ard.add_line(number= 4, name="24p-04c-5v",      voltage=   5, description="", slope=72.0078867850, offset=194.97)
ard.add_line(number= 5, name="08p-2nd-12v",     voltage=  12, description="", slope=69.6503691096, offset=186.98)
ard.add_line(number= 6, name="24p-06c-5v",      voltage=   5, description="", slope=71.2795878411, offset=195.57)
ard.add_line(number= 7, name="08p-1st-12v",     voltage=  12, description="", slope=65.9691784663, offset=192.37)
ard.add_line(number= 8, name="24p-23c-5v",      voltage=   5, description="", slope=71.8246793103, offset=188.22)
ard.add_line(number= 9, name="08p-1st-12v",     voltage=  12, description="", slope=67.0736180009, offset=191.60)
ard.add_line(number=10, name="24p-12,13c-3.3v", voltage= 3.3, description="", slope=71.0547210500, offset=191.25)
ard.add_line(number=11, name="24p-10c-12v",     voltage=  12, description="", slope=68.1271175427, offset=188.17)
ard.add_line(number=12, name="08p-2nd-12v",     voltage=  12, description="", slope=71.5951535529, offset=191.68)
ard.add_line(number=13, name="24p-11c-12v",     voltage=  12, description="", slope=67.8861848384, offset=192.63)
ard.add_line(number=14, name="24p-21c-5v",      voltage=   5, description="", slope=72.1452259359, offset=194.16)
ard.add_line(number=15, name="24p-22c-5v",      voltage=   5, description="", slope=70.4234197519, offset=189.42)
```

A.5. Calibration Script Guide

This short guide explains how the calibration script has to be used:

Start the calibration script `calibration.py`.

```
python calibration.py
Number of samples per ampere value and channel?:
```

Enter the number of times the current values should be sampled. The mean value will be used for linear regression.

```
Ampere value ('q' for quit):
```

Establish the first stable and known current on the electric circuit all channels are connected in series with. Then enter the ampere value read from an amperemeter. Afterwards the measurement will take some time depending on the number of samples and eventually you will be ask to enter the next ampere value. This process repeats for all current values you want to use.

If you finished all current values (at least two) you can type in “q” to quit the calibration. The script will do the linear regression for each channel and provide you a file including the results and all measured data. At the end of the file and also printed to the console a section of code which can be copied and pasted directly into the PMLib settings file can be found.

A.6. Power Measurement Graphs

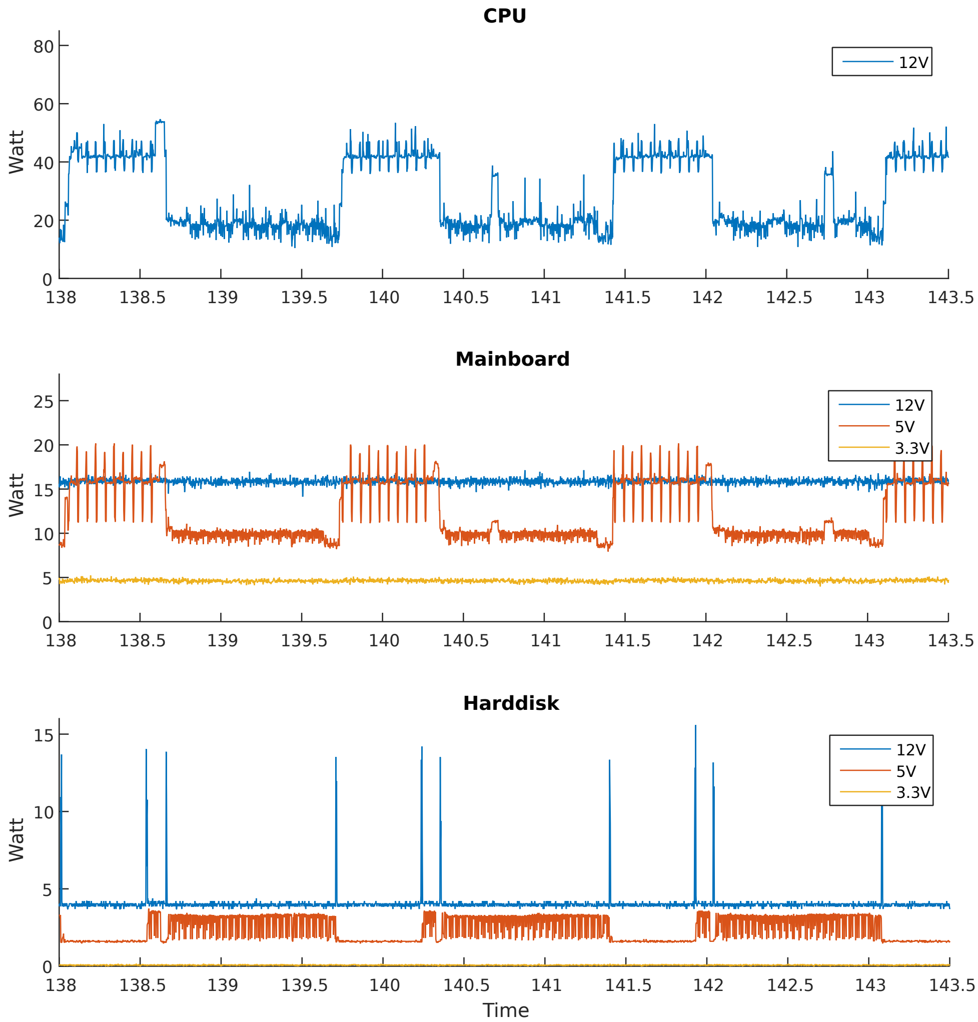

Figure A.6.1: The Partdiff benchmark on one core of the desktop machine.

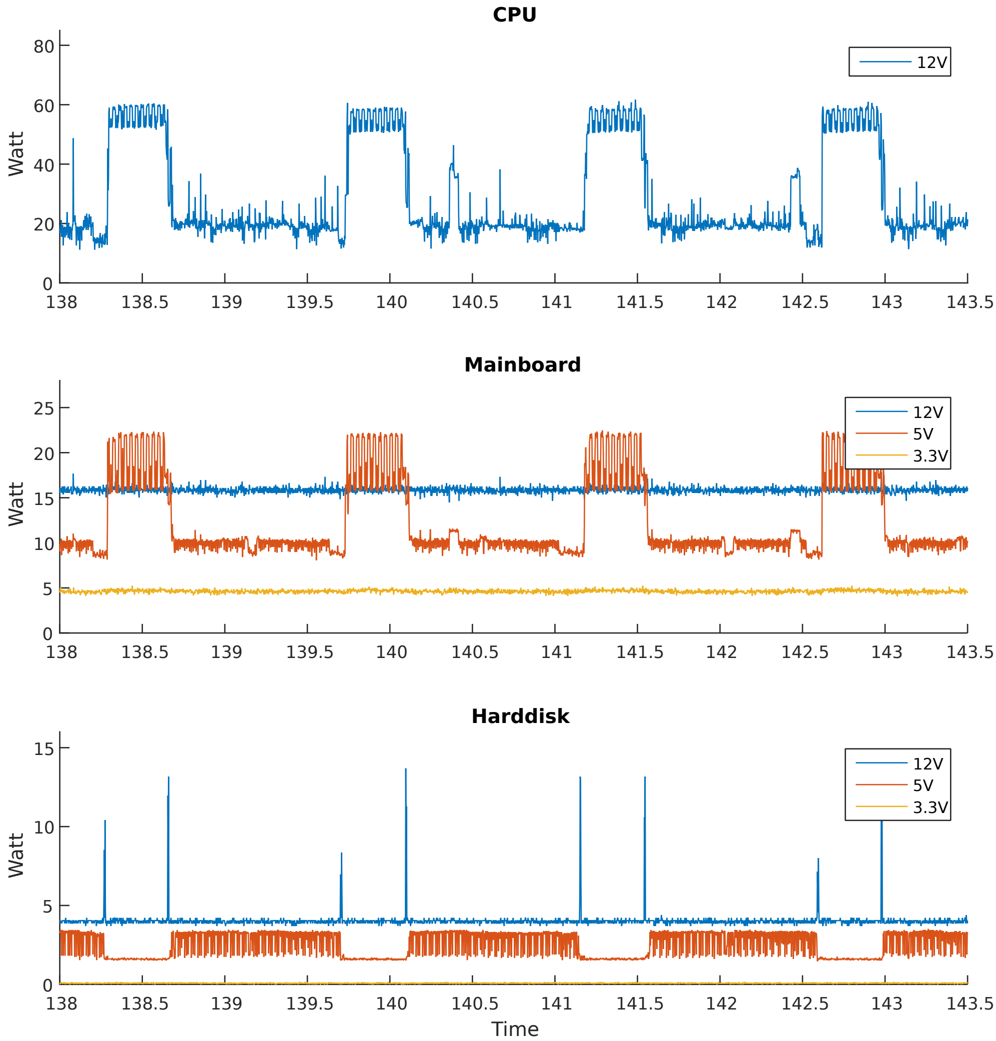

Figure A.6.2: The Partdiff benchmark on two cores of the desktop machine.

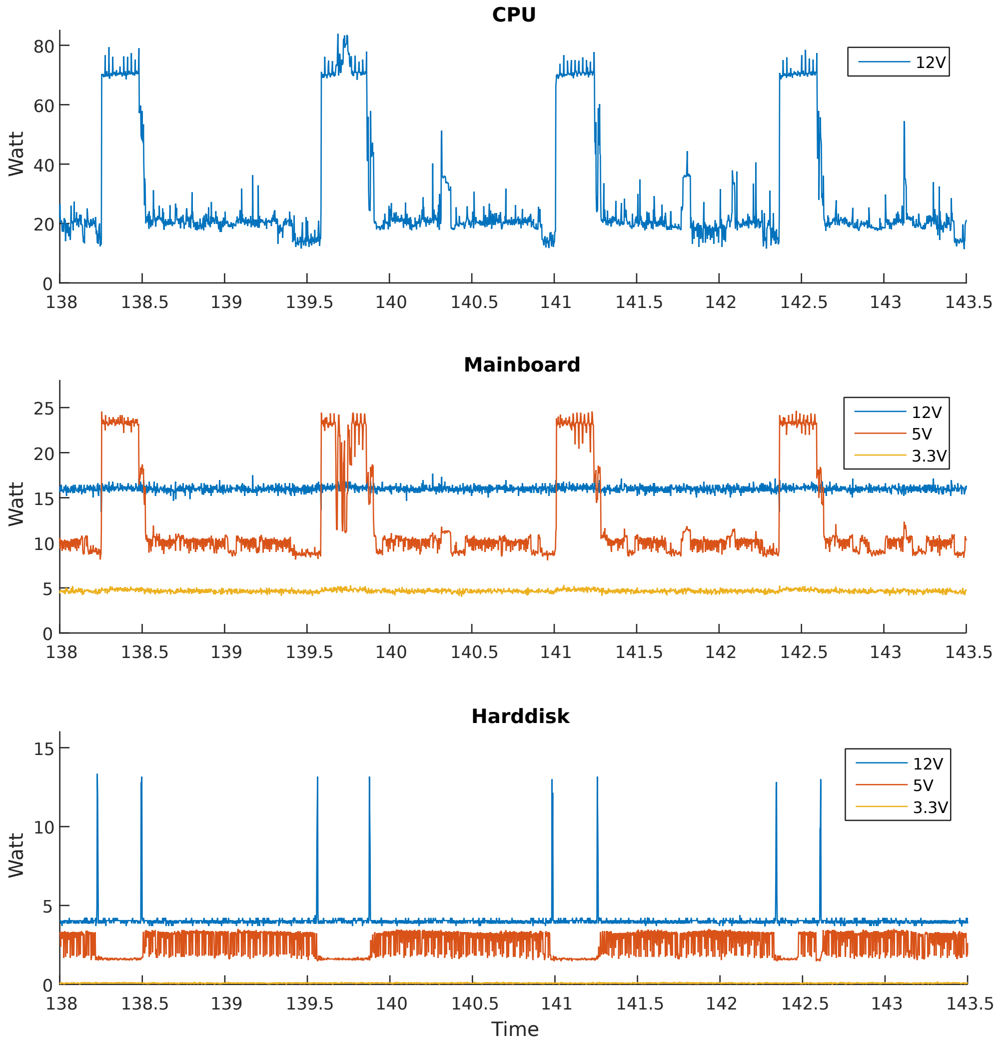

Figure A.6.3: The Partdiff benchmark on all four cores of the desktop machine.

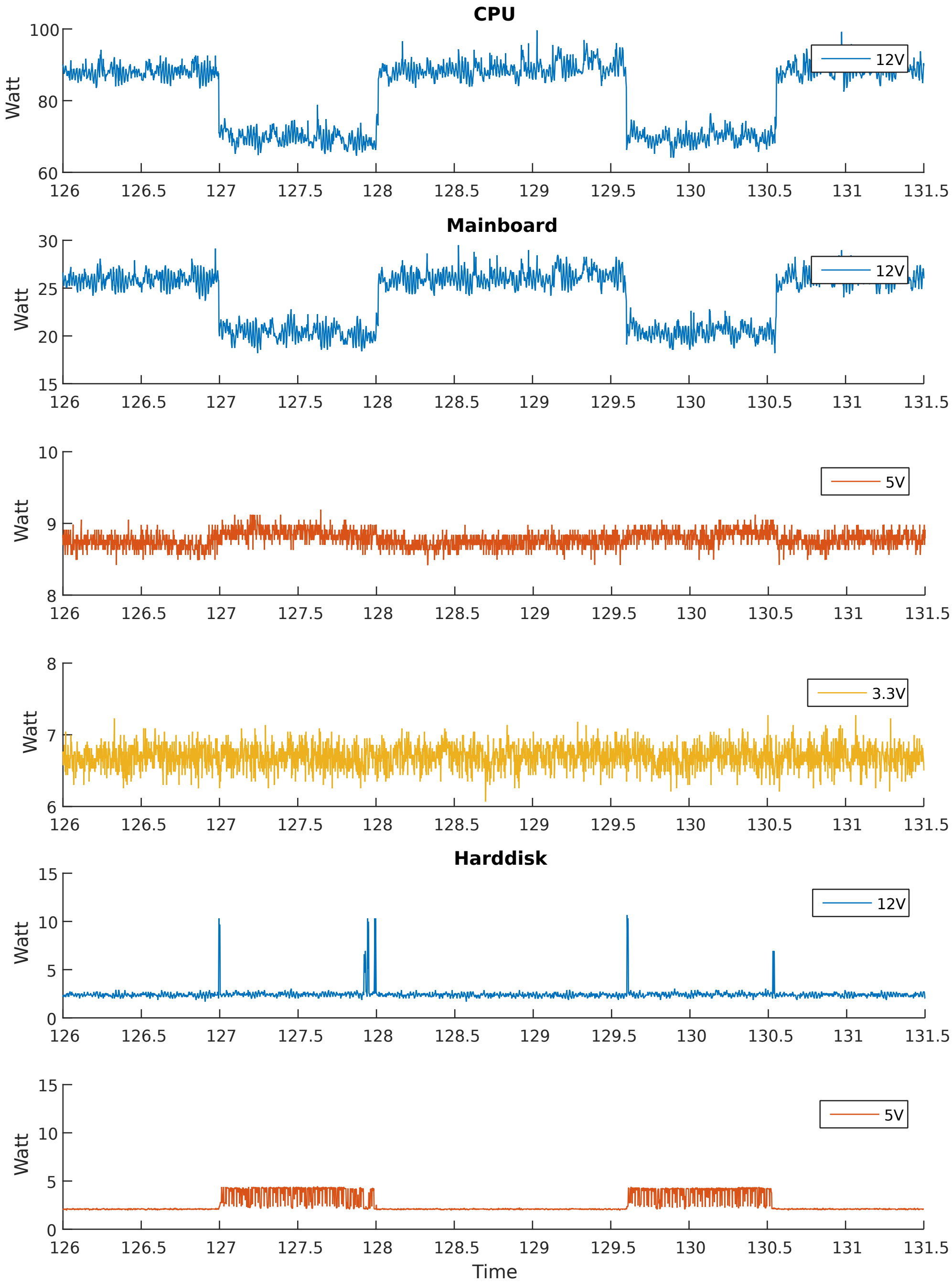

Figure A.6.4: The Partdiff benchmark on one core of the server machine.

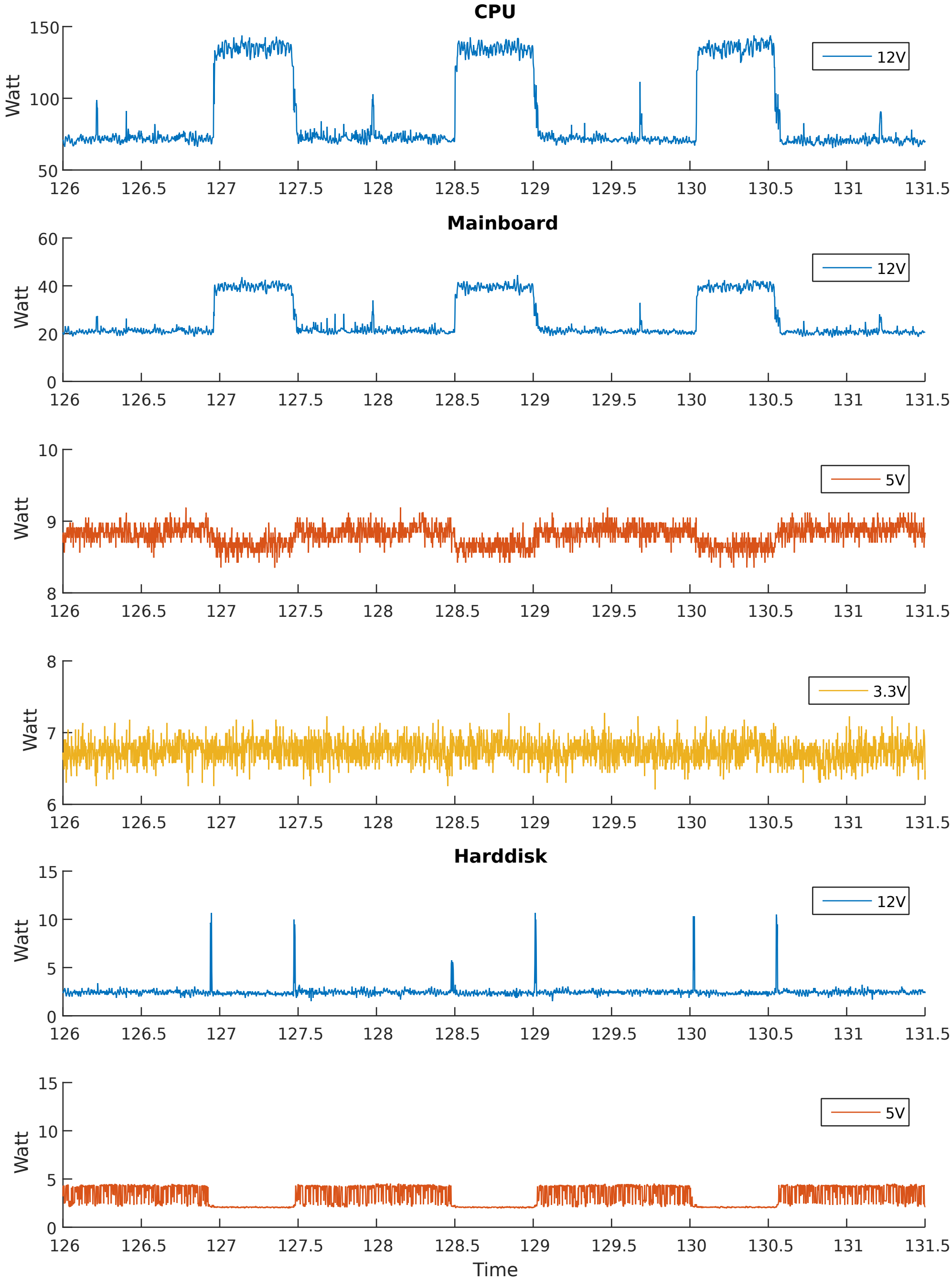

Figure A.6.5: The Partdiff benchmark on four cores of the server machine.

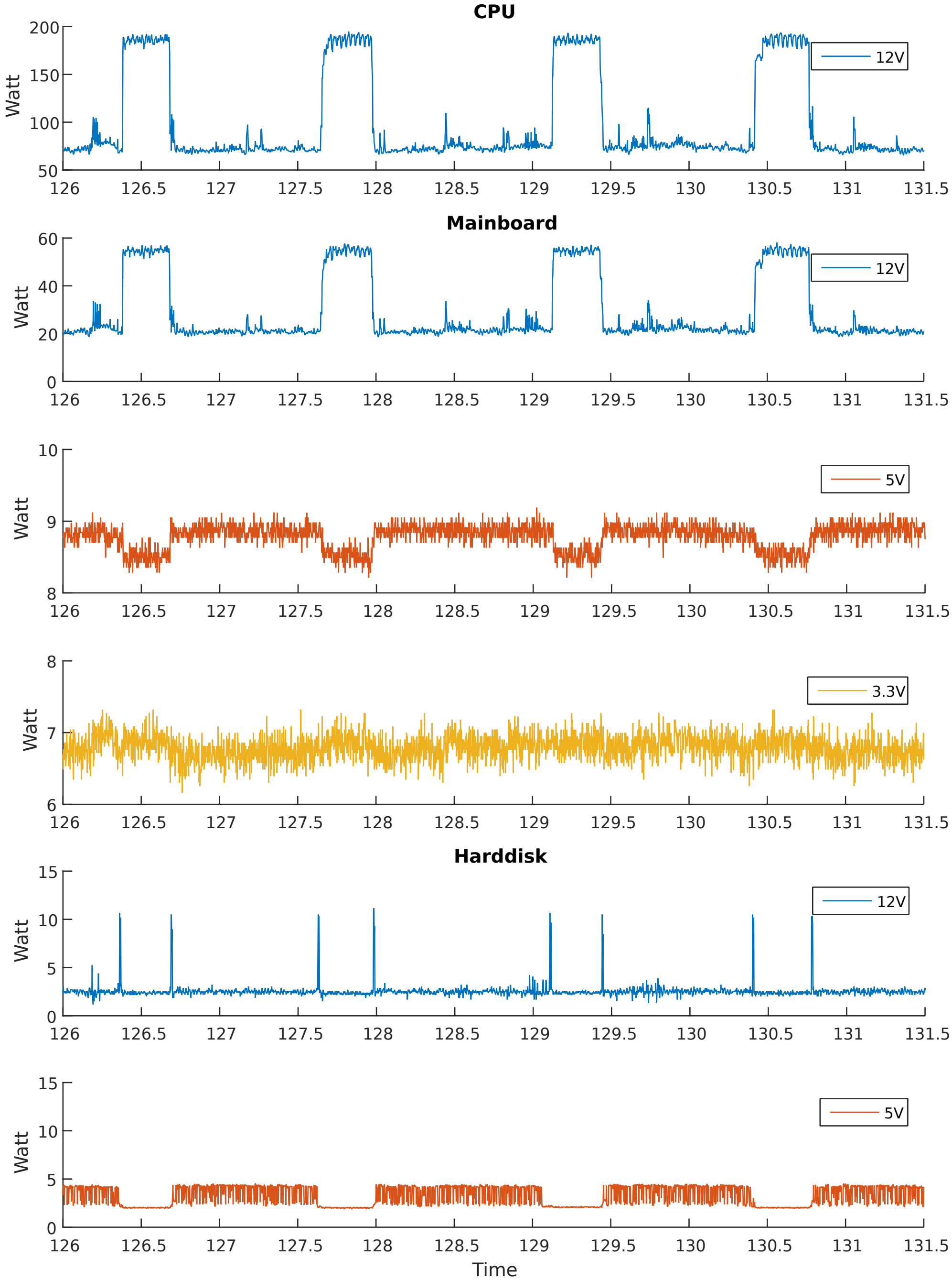

Figure A.6.6: The Partdiff benchmark on all eight cores of the server machine.

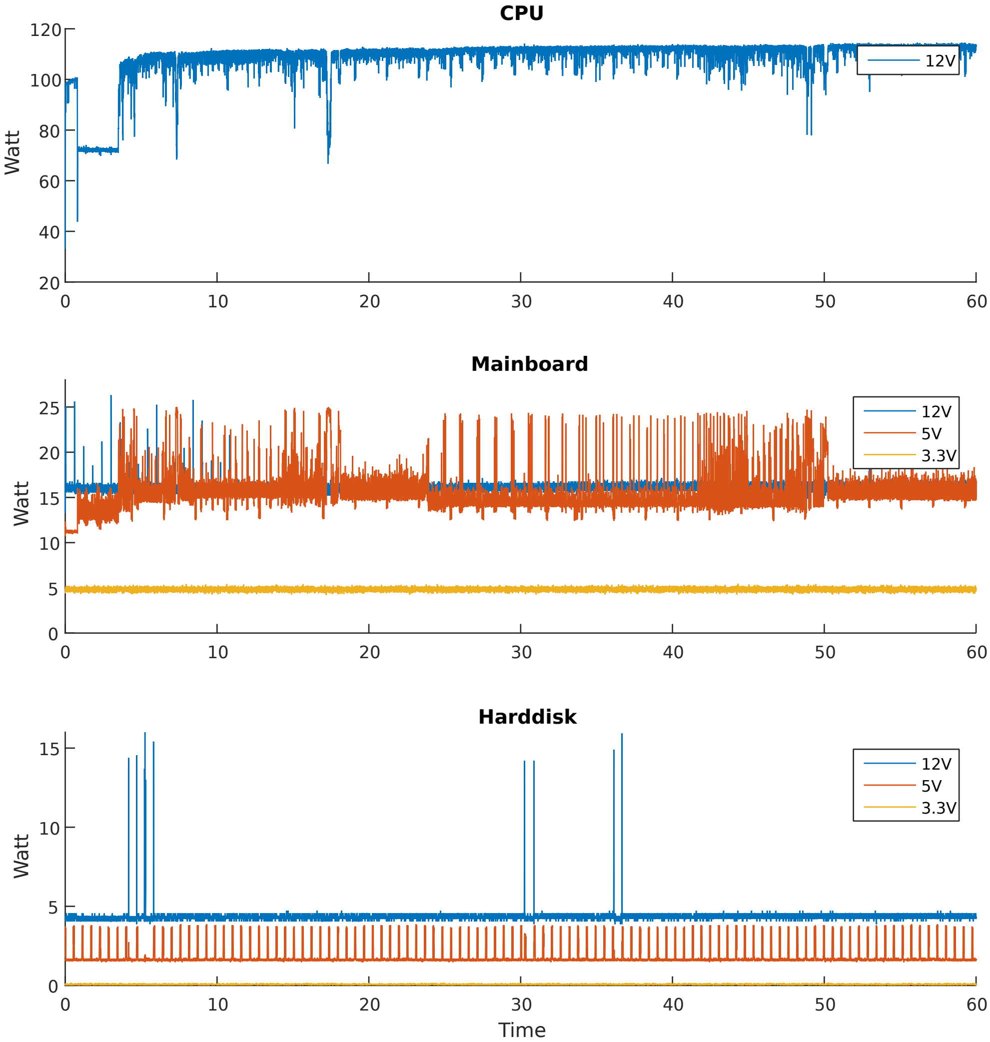

Figure A.6.7: The Linpack benchmark on all four cores of the desktop machine.

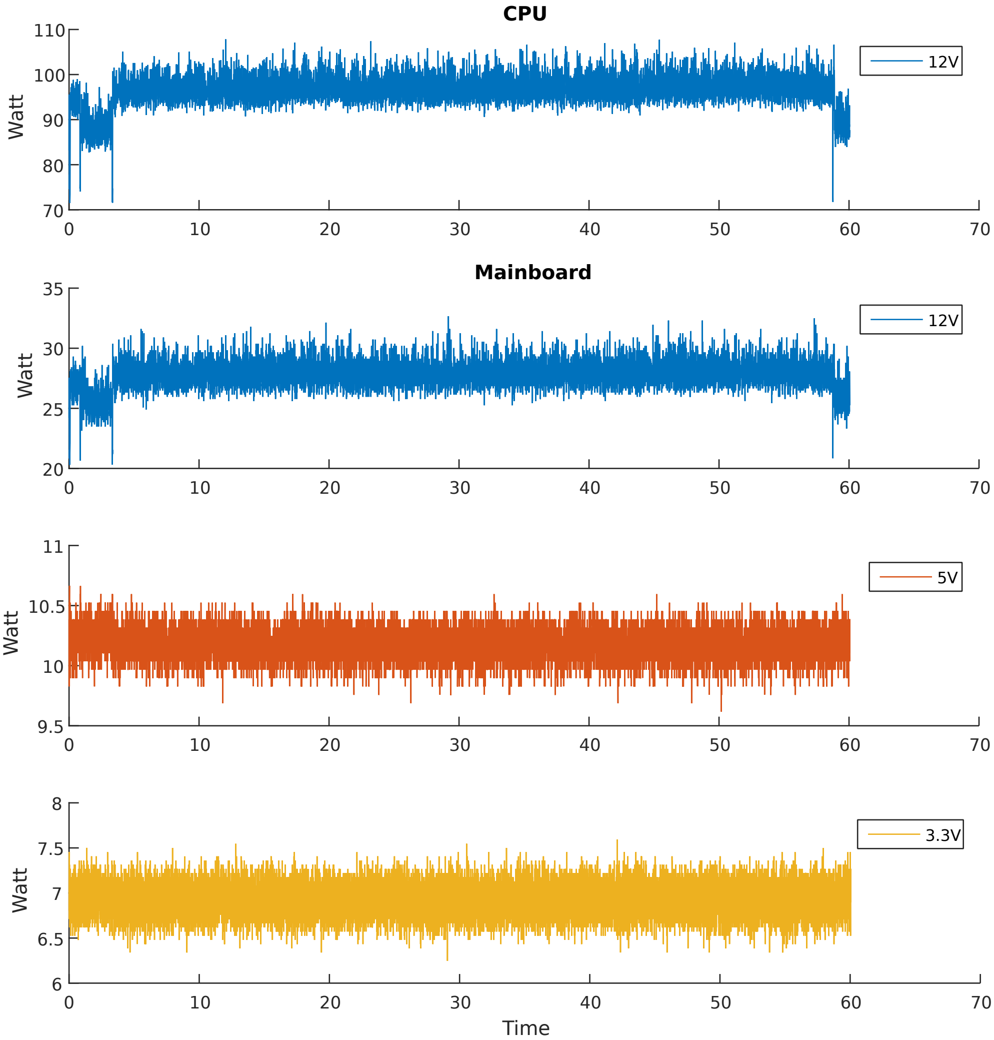

Figure A.6.8: The Linpack benchmark on one core of the server machine.

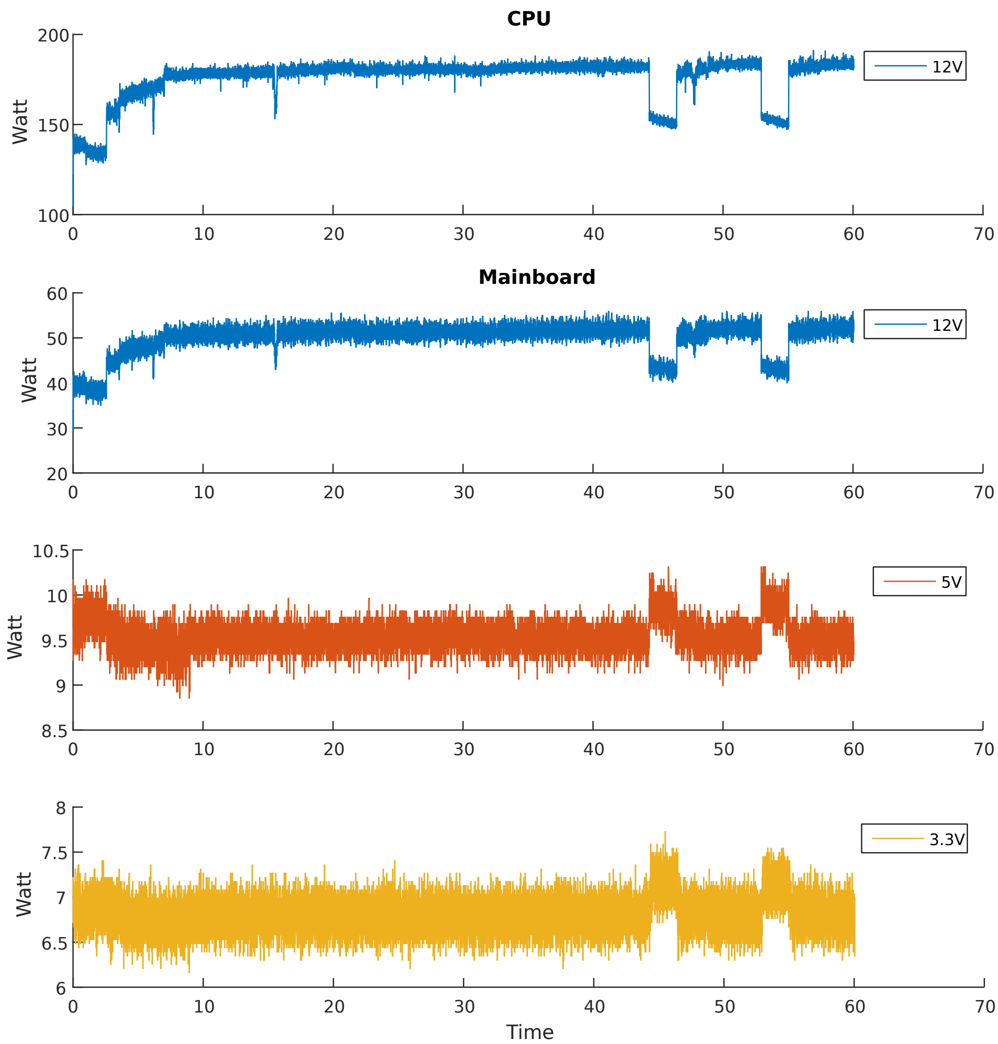

Figure A.6.9: The Linpack benchmark on four cores of the server machine.

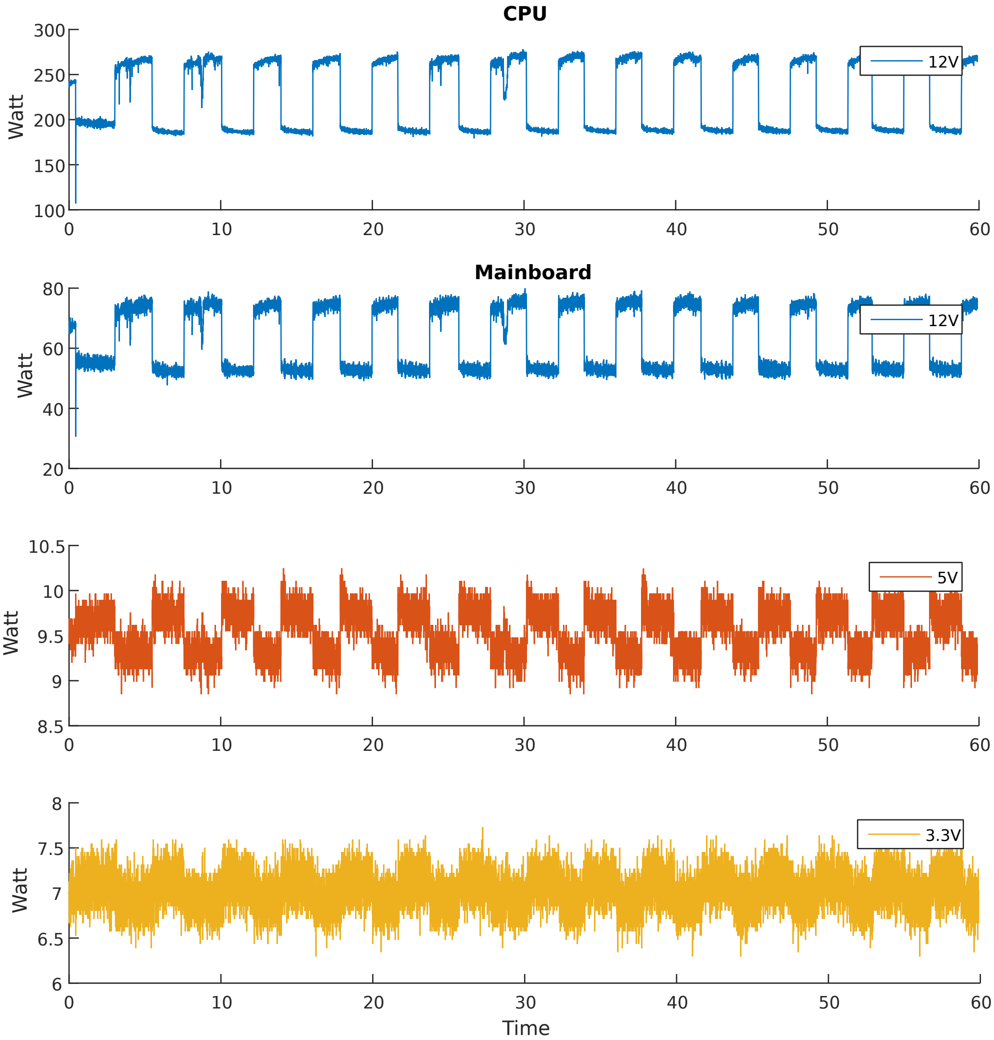

Figure A.6.10: The Linpack benchmark on all eight cores of the server machine.

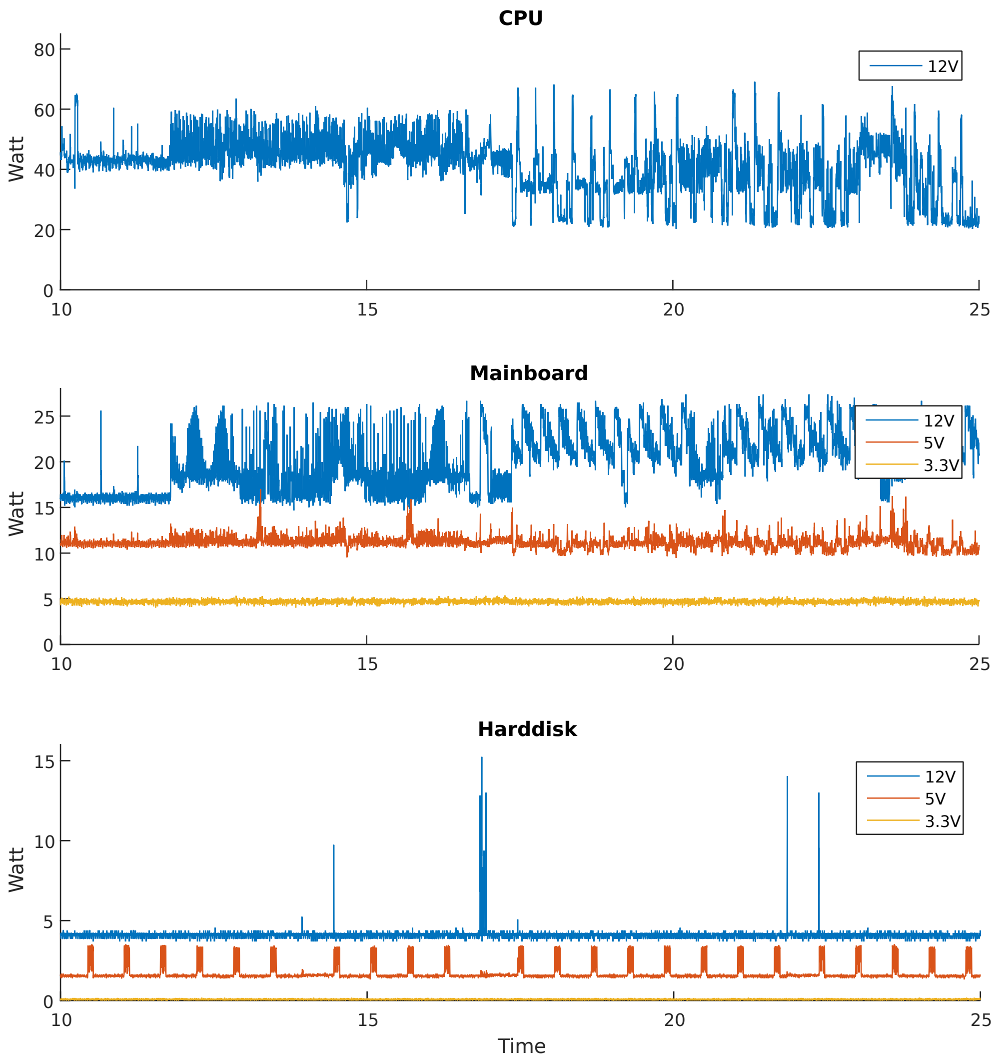

Figure A.6.11: The launch of the Heaven GPU benchmark on the desktop machine.

A.7. Photos of the Setup

Figure A.7.1: The measurement device installed into a desktop machine.

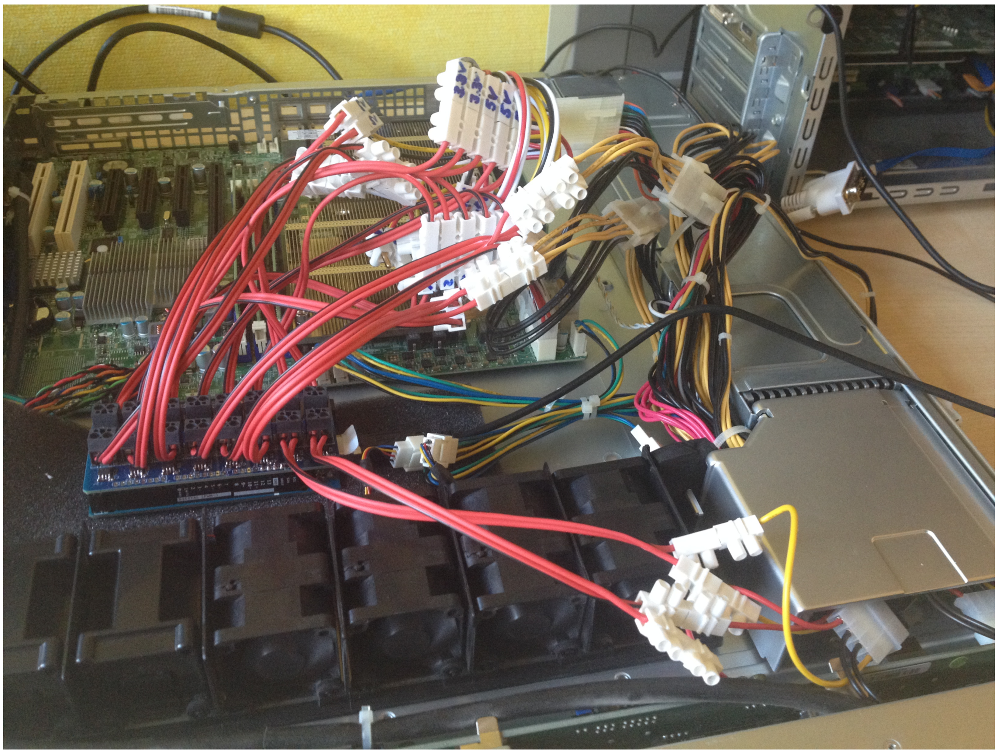

Figure A.7.2: The measurement device installed into a server machine.

References

[1] Lawrence Berkley National Laboratory, *What's Energy Efficiency?*. http://eetd.lbl.gov/ee/ee-1.html, July 13, 2015

[2] Borderstep Institut, *Energieverbrauch und Energiekosten von Servern und Rechenzentren in Deutschland.* http://www.borderstep.de/wp-content/uploads/2014/07/Hintemann-Fichter-Kurzstudie_Rechenzentren_2012.pdf, July 15, 2015

[3] Mahadevan, Priya, et al., *On energy efficiency for enterprise and data center networks.* Communications Magazine, IEEE 49.8 (2011): 94-100.

[4] Hsu, Chung-Hsing, and Stephen W. Poole. *Power measurement for high performance computing: State of the art.* Green Computing Conference and Workshops (IGCC), 2011 International. IEEE, 2011.

[5] Ge, Rong, et al., *Powerpack: Energy profiling and analysis of high-performance systems and applications.* Parallel and Distributed Systems, IEEE Transactions on 21.5 (2010): 658-671.

[6] Bedard, Daniel, et al., *Powermon: Fine-grained and integrated power monitoring for commodity computer systems.* IEEE SoutheastCon 2010 (SoutheastCon), Proceedings of the. IEEE, 2010.

[7] Arduino Website, http://arduino.cc/, April 12, 2015

[8] Hall effect image. Wikipedia. http://upload.wikimedia.org/wikipedia/commons/0/01/Hall_effect.png

[9] ADC image. Wikipedia. http://upload.wikimedia.org/wikipedia/commons/9/9a/Digital.signal.svg

[10] Hall-Effect IC Application Guide. Allegro MicroSystems. http://www.allegromicro.com/en/Design-Center/Technical-Documents/Hall-Effect-Sensor-IC-Publications/Hall-Effect-IC-Application-Guide.aspx

[11] *Power Measurement Library* developed by Michael Kuhn and Manuel Dolz from the work group *Wissenschaftliches Rechnen* at University of Hamburg. https://redmine.wr.informatik.uni-hamburg.de/projects/pmlib

[12] Image from: http://www.conrad.com/ce/en/product/191790/Arduino-A000067-Mega-2560-Microcontroller-Board

[13] Petitet, Whaley, Dongarra, Cleary, *Linpack benchmark.* http://www.netlib.org/benchmark/hpl/, July 19, 2015

[14] John D. McCalpin, *STREAM benchmark.* https://www.cs.virginia.edu/stream/, July 19, 2015

[15] *iPerf*, http://sourceforge.net/projects/iperf/files/, July 19, 2015

[16] Unigine, *Heaven Benchmark*, version 4.0. https://unigine.com/products/heaven/, July 19, 2015

[17] Power Supply Unit. https://en.wikipedia.org/wiki/Power_supply_unit_(computer)

[18] Serial ATA. https://en.wikipedia.org/wiki/Serial_ATA