UNIVERSITY OF CRETE

Building a Power Estimation Model for RISC-V

This thesis is supervised by Dr. Emmanouil Marazakis, Institute of Computer Science, Foundation for Research and Technology - Hellas. The thesis advisor is Prof. Angelos Bilas, Department of Computer Science, University of Crete and Institute of Computer Science, Foundation for Research and Technology - Hellas.

ALEXANDROS TEVRENTZIDIS, University of Crete, Greece

In this project, I constructed an energy consumption estimation model for a *RISC-V* processor using the *SiFive Unmatched* board. External energy measurements as well as internal information provided by the system, such as utilization, were combined in building this model.

Contents

1 INTRODUCTION

1.1 Motivation

The **Power Management Controller** is a system that performs power monitoring and power control of an electronic device. Most computers are built with a set of circuits and software through which the energy consumption of the system can be known at any given time to the rest of the system and, consequently, utilized by it for optimizations at the application level by the kernel (For example, the scheduler may give shorter run times to processes that are seen to consume a lot of energy in order to achieve more economical operation). In the *SiFive* unmatched *board RISC-V*, there is no Hardware Power Monitoring unit, resulting in the system not knowing its energy consumption. This led to the desire to build a model with which energy consumption estimation can be made, information that can be communicated to the system and used for further optimizations at both kernel and user level. Estimating the power by software using a model has advantages beyond its use in systems that do not have a Hardware Power Monitoring unit. An integrated circuit that performs Power Monitoring consists of many electronics. Estimating power using a software model that is applied directly using only information readily accessible to the processor is faster than using a peripheral circuit, which would use complex electronics and converters to perform the same work.
By the term **model**, we mean a parametric algorithm whose parameters are experimentally determined with external measurements combined with information already provided by the system. When there is a model, all that is needed is to be parameterized with this internal information to provide the result.

1.2 Goal

Therefore, the goal of the work is to construct a model that will calculate the energy consumption of the system based on an input, that is, information that the processor has direct access to. This will help in estimating the energy consumption for the specific board and also in finding and developing a methodology for constructing energy models for any system.

2 ENERGY MEASUREMENT METHODOLOGY

To construct the model, I needed to combine power measurements with information already provided by the system. The external measurements represent the actual energy consumed by the processor over a given period during its operation. The internal information is either provided directly by the processor (e.g., performance counters) or by the operating system (e.g., utilization). To calculate the energy consumed by the core, I needed to know the power since energy is the integral of the power.

$$E = \int_{t_0}^{t_1} P(t)\, dt$$

The power P in a circuit component is the product of the current I passing through that point and the voltage V across the two ends of the circuit [Fig1]

$$P_{\mathrm{a,b}} = I \cdot V_{\mathrm{a,b}}$$

The current, in turn, is the ratio of the voltage V to the resistance R of the point for which we want to take the measurement. This is derived from Ohm's Law:

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

Therefore, for a core located between points A and B [Fig2] its power is calculated as follows:

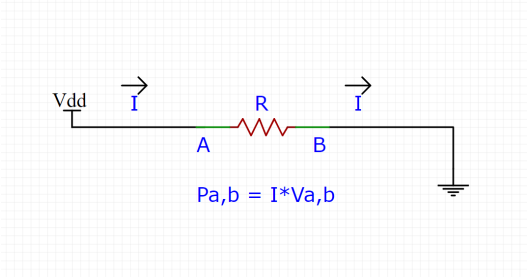


Fig. 1

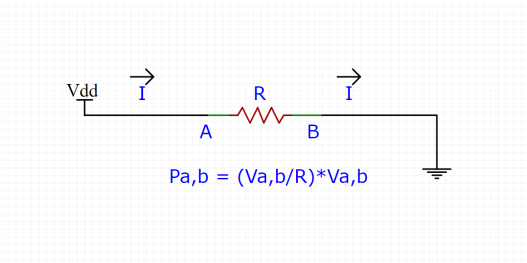


Fig. 2

$$P_{\text{core}} = V_{\text{a,b}} \cdot I_{\text{a,b}} \Longrightarrow P_{\text{core}} = V_{\text{a,b}} \cdot \frac{V_{\text{a,b}}}{R_{\text{core}}}$$

When a core is located between points A and B [Fig3], it is easy to measure the voltage. However, the resistance is indeterminate because the core consists of many dynamic electronic components of which the attributes change over time. The current is not directly computable and thus the power, which now is defined as follows:

$$P(t) = I(t) \cdot V(t)$$

The energy for a time interval $\Delta T_{t_0,t_1}$ is:

$$E(\Delta T) = \int_{t_0}^{t_1} P(t)\, dt$$

To solve this problem, the calculation of the core's power is done in two steps. Initially, we measure the voltage at another point in the circuit where a resistor with a known and constant value is placed [Fig4]. By dividing this voltage across this resistor by its known resistance value, we obtain the current at that point.

The circuit is connected in series. Therefore, according to Kirchhoff's law, the sum of the currents entering a node is equal to the sum of the currents leaving that node. Thus, the current at point A is equal to the current at point B [Fig5]. Therefore, by calculating the current at the point with the known resistance, we indirectly determine the current passing through the core.

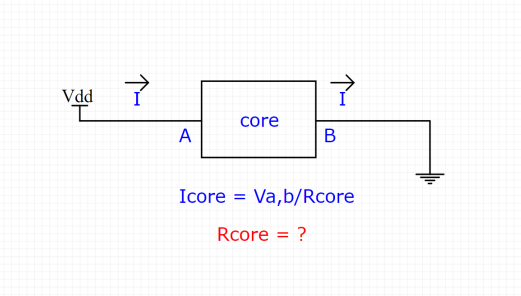


Fig. 3

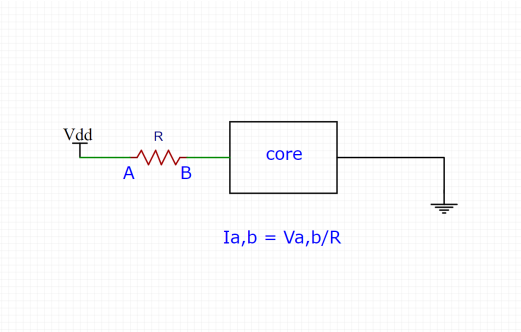


Fig. 4. shunt resistor

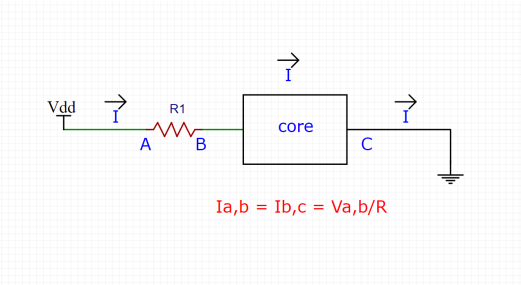


Fig. 5. Kirchhoff's law

This resistance, called a shunt resistor, exists to allow us to measure the current in the core using the above method and has a very small value $1_{\mathrm{m}\Omega}$ so as not to affect the rest of the circuit.

To calculate the power, all that remains is to measure the voltage across the core and then multiply it by the current measured at the point of the shunt resistor [Fig6]. All that is needed is to measure the two voltages, namely the voltage *A*, *B* and the voltage *B*, *C*. This is done using data acquisition devices (*DAQ*). These devices are programmable and typically have a number of analog and digital channels from which they can read and reproduce analog and digital signals, respectively, at a configurable rate, which varies between devices. For their programming, there is usually an API supported by some programming languages.

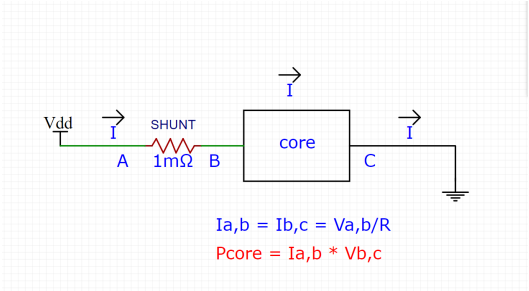


Fig. 6

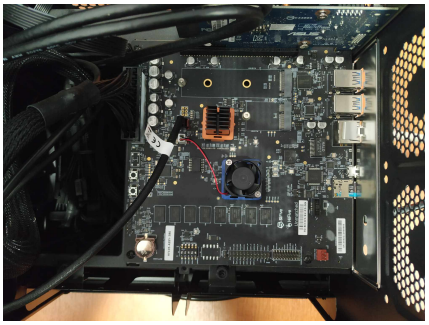


Fig. 7. The board

3 IMPLEMENTATION, HARDWARE

3.1 Board

For my project, I'm using a *SiFive RISC-V Unmatched* board [Fig7] with Gentoo Linux to enable kernel customization and with kernel version 6.6.13. This board has *4 cores*, each with *32KB Instruction Cache* and *32KB Data Cache*. It features *16GB of on-board DDR4 memory* and operates at *1.2 GHz*. Additionally, it provides headers for measuring voltage at the core terminals as well as at both ends of the shunt resistor [4]. Communication with the board is possible via *SSH* and *serial port*.

3.2 DAQ

For voltage measurements, I'm using the *NI USB-3006 DAQ* [Fig8]. This device has 4 analog channels and 13 digital lines, supporting speeds up to 100,000 samples per second, distributed across the 4 analog channels. For device programming, I'm utilizing the *PyDAQmx Python API*. This provides interfaces for programming the channels, both analog and digital, setting the sampling rate, and reading sampled data from the *DAQ* into data structures [3].

3.3 GPIOs

The board also provides 4 *GPIOs* [Fig9] among the rest of the headers. The goal is to use these *GPIOs* to send synchronization signals to the *DAQ*, aiming for greater accuracy and more precise results.

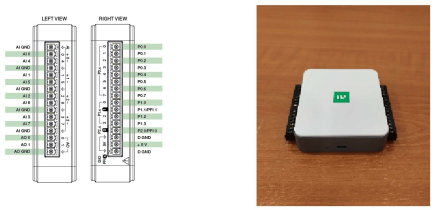


Fig. 8. Daq schematic and picture

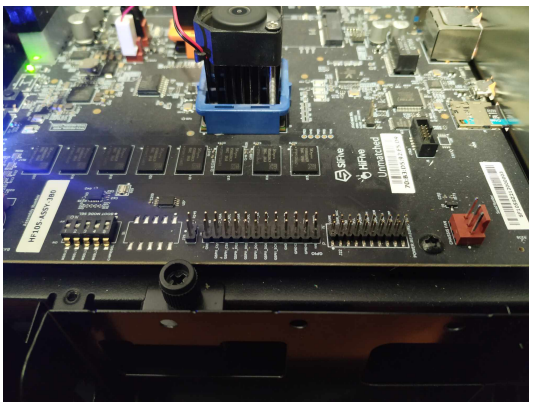


Fig. 9. GPIOs headers

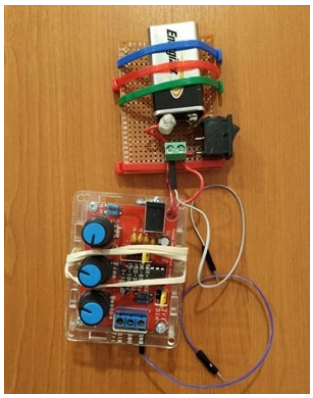
4 IMPLEMENTATION, PROCEDURE

4.1 DAQ testing

For testing the *DAQ* and understanding its operation, I used a function generator [Fig10a] powered by a $9V$ battery as well as two dividers [Fig10b], with ratios of $\frac{1}{100}$ and $\frac{1}{1000}$. I used an analog channel of the *DAQ*, as well as *LabView* software, which visualizes the measurements [2].

4.1.1 Visualize with LabView.

- At 10 volts, the signal was very clean [Fig11].
- Using the $\frac{1}{100}$ divider, the voltage became $\frac{10V}{100} = 100mV$.
  The signal remained clean [Fig12].
- Finally, I attempted to visualize the signal generated by the function generator using a $\frac{1}{1000}$ divider, meaning with a voltage of 10mV. Here, the clarity of the DAQ appeared to be insufficient, resulting in significant noise [Fig13].



(a) Function generator

(b) Dividers

Fig. 10. DAQ testing hardware

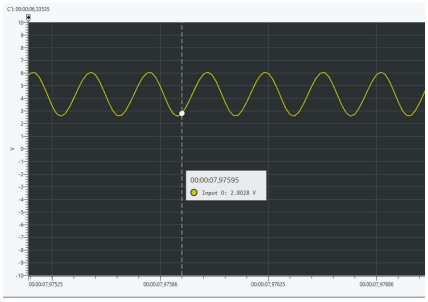


Fig. 11. 10Volt measurement waveform

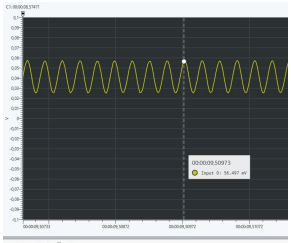


Fig. 12. 100mV measurement waveform

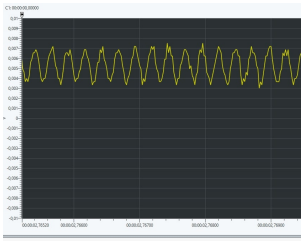
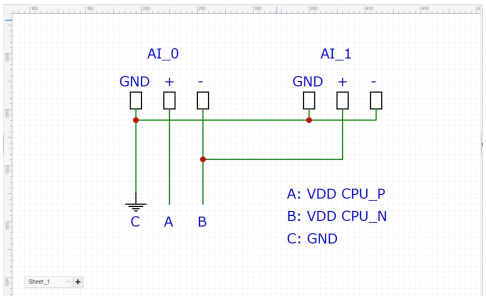


Fig. 13. 10mV measurement waveform

4.2 Connecting the DAQ to the board

The voltage needs to be measured at two points. One is the points A and B on either side of the shunt resistor, and the other is the point B and C on either side of the core [Fig6]. For these measurements, I use the analog channels $AnalogInput_0$ and $AnalogInput_1$, respectively. The connection scheme is shown in Figure14a, while Figure15b depicts the DAQ connected to the headers of the board.



(a) DAQ with headers connection scheme

(b) DAQ connected on headers

Fig. 14. DAQ on board connection

4.3 DAQ programming

To perform the measurements, I first had to program the *DAQ*. The program that handles the *DAQ* to perform voltage measurements is executed on a separate machine, an *Intel i5-4570* running *Windows 10*. This is done to avoid any additional overhead on the cores of the *RISC-V*.

The programming of the *DAQ* is done using the *Python API* provided, and it is divided into four stages.

- Initially, an object of type ***task*** is created to perform the measurements.
- Then the channels used by the task to perform the measurement are configured.
- The rate at which the measurements will be taken (samples per second) is assigned.
- Two callback functions are registered. One is **EveryNSamples**, which, when the task starts, is invoked every time the *DAQ* sends *N* measurements, i.e., once per second. The other callback function is **Done**, which is called when the task finishes in order to perform the necessary cleanups.

The **EveryNSamples()** function runs on a thread that is called every time the *DAQ* sends data and starts as soon as the task is started. To stop this function from being executing, the whole task must be terminated.

An implementation like the following has the problem that the **StartTask()**, which is a **non** blocking function, will be called, which will start the thread, but immediately the **StopTask()** will also be called, which will terminate the sampling thread, and thus the whole sampling process will be terminated immediately.

```
1: procedure SAMPLE                ▷ The sampling process here is terminated immediately
2:     STARTTASK()                 ▷ Starts the EveryNSamples thread
3:     STOPTASK()                  ▷ Stops the EveryNSamples thread
4: end procedure
```

For this reason, a stall mechanism between the 2 routines is necessary. This mechanism will stall the program letting the **EveryNSamples** execute until a signal will trigger the blocking mechanism to resume the execution flow. A simple stall mechanism is *input()* system call which waits for an input.

```
1: procedure SAMPLE                    ▷ The sampling process here waits for a signal to stop
2:     STARTTASK()                     ▷ Starts the EveryNSamples thread
3:     INPUT()                         ▷ Stall here, press return to resume
4:     STOPTASK()                      ▷ Stops the EveryNSamples thread
5: end procedure
```

4.4 Samples storing

The two voltages measured by the *DAQ* are stored in a *CSV* file with two columns.

- One column for the voltage at points *A* and *B* (shunt)
- One column for the voltage at points *B* and *C* (core)

At a later time, I read this *CSV* file, and for each pair of voltages, I calculate the corresponding current and power using the known value of the shunt resistor in the manner analyzed in Chapter2. The calculated power then is stored in the same *CSV* file by creating two additional columns: one for the current computed by the particular voltage pair and one for their corresponding power.

4.5 DAQ with GPIOs

The *DAQ* has 13 digital lines, which can read and send digital signals. Two of these lines are called *PFI* and can be used as triggers. By properly configuring them with the API, they can initiate an analog sampling task as soon as they read a high signal.

To make the measurements more precise, I thought of using 3 out of the 4 *GPIOs* of the board for the following purpose each.

- **Start Signal**:A high logic *GPIO* that sends a sampling **start signal** to the *DAQ*.
- **Stop Signal**: A high logic *GPIO* that sends a sampling **stop signal** to the *DAQ*.
- **Filtering Signal**: It may be useful to apply some filtering during sampling to ignore values that are not relevant for the particular experiment. The filtering *GPIO* can oscillate between high and low during sampling and, in the end, only retains the samples that were taken with a high **Filtering Signal**.

The above also require appropriate programming of the *DAQ*. These are digital signals, so I need to use the digital lines of the device.

- **Start signal**
  The *DAQ NI USB-6003* provides 2 *PFI* digital lines that can be used as start triggers. Specifically, I used PFI_0 to connect the *DAQ* with the *GPIO start signal*.

  SetStartTrigger(task, PFI_0***);***

  Both *PFI* are edge-triggered. As soon as the configured one reads high, it starts the task automatically and initiates the **EveryNSamples()** thread.

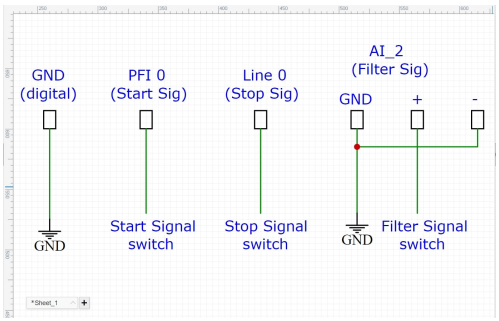
- **Stop signal**
  The *DAQ NI USB-6003* does not have the ability to stop a task via digital triggering as it does for starting one. To implement the **Stop Signal**, I used a digital line, specifically $line_0$, and constructed a mechanism that creates a thread performing polling on the value of this digital line.

```
1: procedure SAMPLE                                  ▷ GPIOs instruct the sampling process
2:     POLLING.START()                               ▷ Starts polling of digital line_0
3:     STARTTASK()                                   ▷ Starts sampling when PFI_0 is triggered
4:     POLLING.SPINONBIT(1)                          ▷ Spins on digital line_0 while its value is high
5:     POLLING.STOP()                                ▷ When value of line_0 becomes low, stalling stops
6:     POLLING.JOIN()
7:     STOPTASK()                                    ▷ StopTask is called to stop the sampling
8: end procedure
```

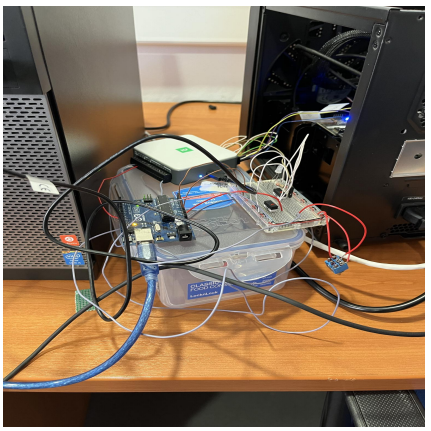
- **Filtering signal**
  The **Filtering Signal** is a signal that goes high when useful values for the experiment are being sampled. Therefore, polling of the **Filtering Signal** and analog sampling must be performed simultaneously. Another issue with the *DAQ NI USB-6003* is that analog and digital measurements cannot be synchronized. For this reason, the **Filtering Signal** is read from an analog channel of the *DAQ* rather than a digital one. Voltage signals greater than 3 volts are considered high, while the rest are considered low. These signals are stored in a buffer parallel to the one storing the samples. At the end of sampling, the two parallel buffers are multiplied element-wise, thereby retaining only the measurements for which the Filtering bit was high.

4.6 GPIOs Simulation

To test the functionality of the *DAQ* with the *GPIOs*, I used an ***Arduino Uno*** [1] to send digital signals to the 3 inputs (**Start/Stop/Filter**) using switches and I connected these switches with the digital lines of the *DAQ* [Fig15].



(a) GPIO to DAQ schematic



(b) Sampling with simulated GPIOs

Fig. 15. GPIO signal sampling

4.7 Building the model

To construct the model, I take measurements while a benchmark is running and simultaneously record the core utilization in a trace file at a constant and predefined rate.

Dhrystone

The benchmark I used to construct the model is *Dhrystone*. *Dhrystone* is a tight loop benchmark that gradually spawns processor threads to utilize the core. It is configurable in terms of the number of threads it will use as well as the duration between the start of one thread and the start of the next.

The variation of *Dhrystone* I use allows me to record the core utilization at a configurable rate in a trace file by reading the output of the */proc/stat* file while the whole benchmark runs.

Having a *CSV* file with the power measured during the execution of *Dhrystone* and a file with the core utilization per 1 second, I created a scatter plot with the utilization on the x-axis and the average power corresponding to that utilization on the y-axis [Fig16]. Plotting the power column reveals that it increases with the initiation of a new thread. To achieve a clearer picture due to the abundance of power data, I calculated the rolling mean with a window of 200 [Fig17].

Scatter plot (x: utilization, y: power)

The data of the power measured in the system as well as the utilization traced while *Dhrystone* was running were mapped and plotted [Fig18].
The data shows linearity [Fig18], so I needed to apply linear regression to construct a linear model [Fig19]. The resulting line is of the form $y = a \cdot x + b$, where x represents the utilization. Specifically, the line equation is

$$0.002912 \cdot x + 3.561$$

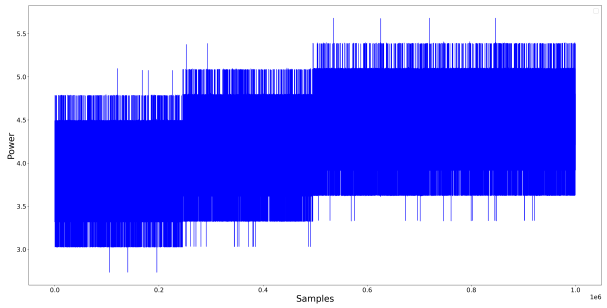


Fig. 16. Power plot over time

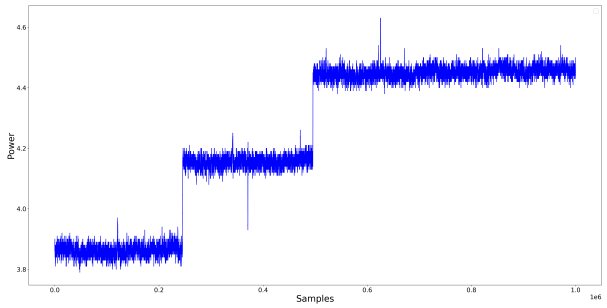


Fig. 17. Power rolling mean plot, win_size: 200

- The variable $\boldsymbol{x}$ takes the values of utilization, which is measured in ***Jiffy*** units.
- The parameter $\boldsymbol{a}$ is the multiplicative term of the model, indicating how much power increases as utilization increases. It is measured in $\frac{Watt}{Jiffy}$.
- The parameter $\boldsymbol{b}$ the constant term of the equation of the line, measured in Watt units. This model predicts the system's energy given utilization. By placing the values of utilization in $\boldsymbol{x}$, power can be predicted in ***Watt***.

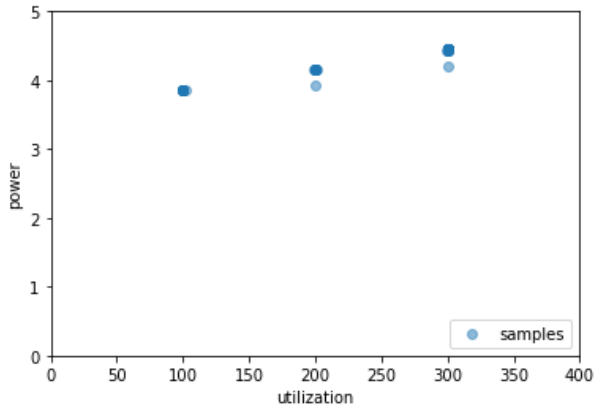


Fig. 18. power, utilization plot

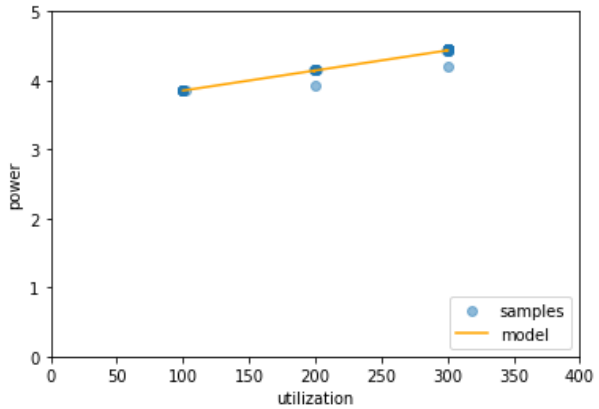


Fig. 19. Linear regression

Should this model be parameterized with the utilization, it will compute the energy consumed by the system. By placing the values of utilization in $\boldsymbol{x}$, power can be predicted in ***Watts***.

$$\textbf{PowerEstimate} = \mathbf{0.002912} \cdot \mathbf{x} + \mathbf{3.561}$$

5 VALIDATION WITH STREAM

To validate the model, I used a second benchmark, *STREAM*. *STREAM* is divided into 4 phases, which are executed sequentially for ***NTIMES*** times. The four phases are:

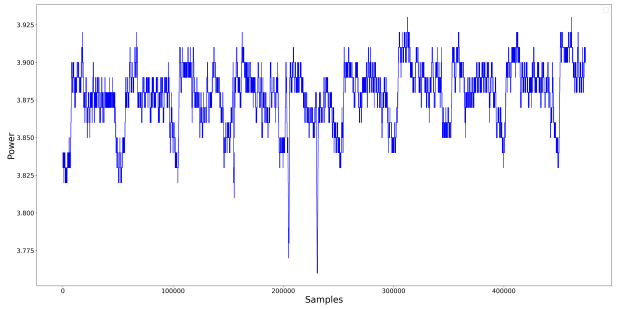


Fig. 20. STREAM power rolling mean, win_size: 1000

(1) Copy
(2) Scale
(3) Add
(4) Triad

Each phase is executed in parallel using the number of cores available for execution. Each phase is applied to the elements of an array with a configurable size set at compile time using the parameter *STREAM_ARRAY_SIZE*.

For *STREAM*, as well as for *Dhrystone*, I utilized 3 out of the 4 cores to execute the benchmark, reserving one core to snapshot the */proc/stat* output in order to collect utilization data with a rate of $\frac{1snapshot}{100ms}$. Additionally, I execute STREAM with *STREAM_ARRAY_SIZE* set to 120,000,000 elements so to ensure an execution period enough for a number of samples to be generated for the *DAQ*.

I initiated *STREAM* with *NTIMES = 10* and simultaneously started sampling with the *DAQ*. Then I plotted the measured power [Fig20].

In plot [Fig20] it can be seen that there is a periodicity observed due to the fact that each set of four phases runs *NTIME* times, so every period referes to a *NTIME*. Additionally, it appears that during each period, the utilization reaches maximum values at the beginning, namely during the *Copy* phase.

5.1 Validation

I used the model derived from *Dhrystone* and, at this particular model, I applied the utilization values obtained while executing *STREAM* to predict the power. Then, I compared the predicted power with the actual power values sampled while *STREAM* was running.

The diagram [Fig21] represents the predicted power by applying the traced utilization during *STREAM* execution to the model constructed with *Dhrystone*.

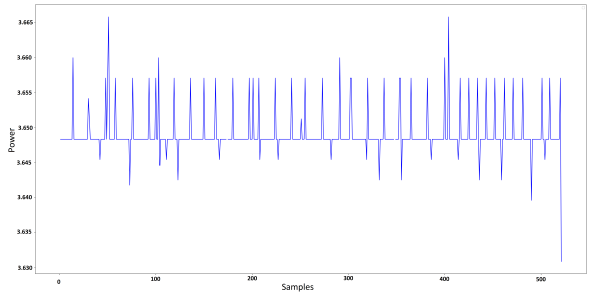


Fig. 21. Predicted power, with model parameterized with STREAM utilization values

- Computing the mean squared error, I obtained the value 0.05058280972320814.
- Computing the root mean squared error, I obtained the value 0.22490622428738635.
- Computing the normalized root mean squared error, I obtained the value 0.0580666562884.

This means there is a prediction error of 5.8%.

6 CONCLUSION

In this project, I used a *RISC-V SiFive Unmatched* board to construct an energy estimation model using both external measurements and internal information. The external measurements included the voltage at both ends of the core [Fig6] and the shunt resistor to calculate the energy consumed by the core. These measurements were taken using the *DAQ NI USB-6003*. The internal information used to build the model was the system's utilization, which was obtained by reading the file **/proc/stat** periodically.

For constructing the model, I used the *Dhrystone* benchmark to utilize the core by waking up its threads. The *DAQ* sampled the voltage at the points *A*, *B* and *C*, *D* [Fig4] while the *Dhrystone* was running, and simultaneously, the system kept track of the core utilization in a trace file. By correlating the utilization values with the corresponding power values measured and, in a second time, calculated using the *DAQ*, I constructed the model $0.002912 \cdot x + 3.5610.002912 \cdot x + 3.561$. This model predicts the system power in watts when the utilization of the system is substituted for the variable x. The unit of measurement for utilization is *Jiffy*. Parameter a indicates the rate of change of power relative to utilization and has the unit $\frac{Watt}{Jiffy}$, while parameter b is the constant term of the equation and has the unit *Watt*.

To validate the model, I used a second benchmark, *STREAM*, following the same procedure. The *DAQ* sampled during the execution of *STREAM*, and one thread stored the core utilization at regular time intervals. Using these utilization values and applying them to the model, I predicted the power. I compared the predicted power values with the actual power values measured using the *DAQ*. The root mean squared error was calculated to be **0.22490622428738635**, and after normalization, it was **0.0580666562884**. Thus, the error is approximately **5.8%**.

Despite the model being computed using minimal information (only the utilization), the error percentage is quite small. This is because the parameter b of the linear model (with a value of 3.561) is relatively large. In other words, according to the linear model, the system power will always be at least 3.561 watts, even when the utilization is zero.

The calculated error is small; however, it likely stems from the fact that the *RISC-V SiFive Unmatched* is a relatively simplistic system. In larger and more complex systems, utilization information may not be sufficient, and a linear model may be inaccurate.

7 FUTURE PLANS

7.1 Performance counters

Using more information provided by the system will improve the accuracy of the model, especially in the case of more complex systems. Specifically, using performance counters with the help of the *perf* program and combining them with utilization can help recalculate the model more precisely. Additionally, obtaining more accurate results by measuring power while the system is idle (idle power), subtracting this value, and predicting non-idle power can also be beneficial.

7.2 GPIOs

To ensure that the *DAQ* measurements are synchronized and taken only at necessary moments, it is advantageous to use 3 *GPIOs* of the board connected to the *DAQ* as previously described in the Implementation chapter [Section4.5]. This work requires utilizing the system's *GPIOs* drivers.

7.3 Amplifier

Finally, the ina128p amplifier will be placed on the board's headers to amplify the signal, resulting in more accurate *DAQ* measurements with less noise [5].

7.4 Static Power

The constant term b in the model equation is very likely to represent the static power of the system, which is observed when the utilization is zero. However, this claim needs to be substantiated, and I need to determine with greater precision whether b indeed represents the static power and which part of it is the static power by clarifying initially what the *RISC-V* idle state means is determined.

REFERENCES

[1] Arduino. *Arduino Uno.* Arduino, 2024. https://www.arduino.cc/en/Main/ArduinoBoardUno.
[2] National Instruments. *LabVIEW.* National Instruments, 2024. https://www.ni.com/en-us/shop/labview.html.
[3] National Instruments. *NI USB-6003 Data Acquisition Device.* National Instruments, 2024. https://www.ni.com/en-us/support/model.usb-6003.html.
[4] SiFive. *SiFive HiFive Unmatched RISC-V Development Board.* SiFive, 2024. https://www.sifive.com/boards/hifive-unmatched.
[5] Texas Instruments. *INA128P Precision Instrumentation Amplifier.* Texas Instruments, 2024. http://www.ti.com/product/INA128.