Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

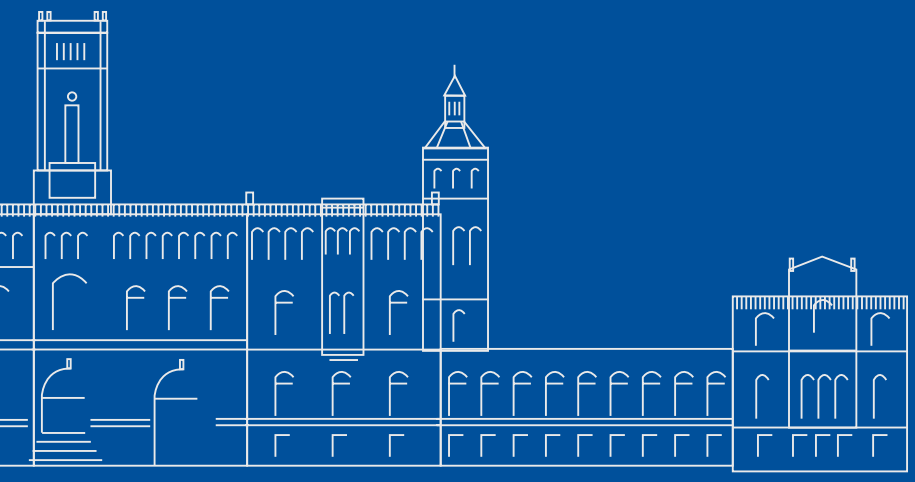
Johannes Karl Arnold

On the Power Estimation of a RISC-V Platform using Performance Monitoring Counters and RTOS Events

Bachelorarbeit im Fach Informatik

6. November 2024

Please cite as:
Johannes Karl Arnold, "On the Power Estimation of a RISC-V Platform using Performance Monitoring Counters and RTOS Events" Bachelor's Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, November 2024.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

On the Power Estimation of a RISC-V Platform using Performance Monitoring Counters and RTOS Events

Bachelorarbeit im Fach Informatik

vorgelegt von

Johannes Karl Arnold

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr. Jan Simon Rellermeyer**
Betreuer:	**Tim-Marek Thomas, M.Sc.**

Beginn der Arbeit:	**12. Juni 2024**
Abgabe der Arbeit:	**14. Oktober 2024**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Johannes Karl Arnold)
Hannover, 6. November 2024

ABSTRACT

With the advent of the “zettabyte era” in the mid-2010s, power consumption has become an increasing topic of interest as the number of computer systems continues to rise, affecting large datacenters and consumer grade devices as well as embedded systems. Energy monitoring and estimation has a significant impact on a number of key areas, including compiler optimization, scheduling, thermal- and battery life management, as well as potential long-term economical and environmental consequences.

While many contemporary CISC platforms incorporate features such as RAPL to estimate power consumption, estimating the power consumed by a RISC processor often presents a greater challenge in the absence of specialized hardware extensions, particularly in the context of embedded systems.

This thesis examines the time and power consumption characteristics of a common embedded RISC-V processor using a diverse set of algorithms representative of an embedded system. It employs a bespoke benchmarking framework designed around the collection of PMC data. The data is then subjected to analysis and transformation, and used to train and evaluate a generalized model, thereby enabling the prediction of the system’s power consumption from PMC data alone.

The final model was able to predict the SoC’s current draw with an error of around 0.88 % when using data from benchmarks it was not trained on. This outcome provides compelling evidence that PMC data can be effectively employed for the aforementioned use cases. The correlations identified from PMC benchmarking data are then aligned with the tracing framework of a contemporary RTOS, which could also benefit from run-time energy statistics.

KURZFASSUNG

Mit dem Beginn der „Zettabyte-Ära“ Mitte der 2010er Jahre ist der Stromverbrauch zu einem immer wichtigeren Thema geworden wie die Zahl der Computersysteme weiter steigt, sowohl in großen Rechenzentren als auch bei Verbraucherelektronik und eingebetteten Systemen. Die Überwachung und Abschätzung des Energieverbrauchs hat erhebliche Auswirkungen auf eine Reihe von Schlüsselbereichen, darunter Compiler-Optimierung, Scheduling, Wärme- und Batterielebensdauer-Management sowie potenzielle langfristige wirtschaftliche und ökologische Folgen.

Während viele moderne CISC-Plattformen Funktionen wie RAPL zur Abschätzung der elektrischen Leistung enthalten, stellt die Abschätzung der Leistung eines RISC-Prozessors oft eine größere Herausforderung dar, da es wenig bis keiner speziellen Hardware-Erweiterungen gibt, insbesondere bei eingebetteten Systemen.

In dieser Arbeit werden die Zeit- und Stromverbrauchseigenschaften eines gewöhnlichen eingebetteten RISC-V-Prozessors unter Verwendung einer Reihe von Algorithmen untersucht, die für ein eingebettetes System repräsentativ sind. Es wird ein maßgeschneidertes Benchmarking-Framework verwendet, welches für die Sammlung von PMC-Daten entwickelt wurde. Die Daten werden dann einer Analyse und Transformation unterzogen und zum Trainieren und Bewerten eines verallgemeinerten Modells verwendet, wodurch die Vorhersage des Stromverbrauchs des Systems allein anhand der PMC-Daten ermöglicht wird.

Das endgültige Modell war in der Lage, die Stromaufnahme des SoC mit einem Fehler von etwa 0.88 % vorherzusagen, wenn Daten von Benchmarks verwendet wurden, für die es nicht trainiert wurde. Dieses Ergebnis ist ein überzeugender Beweis dafür, dass PMC-Daten für die oben genannten Anwendungsfälle effektiv genutzt werden können. Die aus den PMC-Benchmark-Daten ermittelten Korrelationen werden dann mit dem Tracing-Framework eines modernen RTOS abgeglichen, das ebenfalls von Laufzeit-Energiestatistiken profitieren könnte.

Es ist unwürdig, die Zeit von hervorragenden Leuten mit knechtischen Rechenarbeiten zu verschwenden, weil bei Einsatz einer Maschine auch der Einfältigste die Ergebnisse sicher hinschreiben kann.

Gottfried Wilhelm Leibniz

CONTENTS

1 INTRODUCTION

Small computer systems have permeated nearly all parts of life in the 21st century, and it is predicted that by 2035, these embedded and IoT systems will cumulatively surpass one trillion in number, spending many years in service having an extraordinary impact on digital infrastructure [Spa17]. As the sheer number of use cases for these embedded systems continues to rise drastically, so has the interest in their power consumption.

Ensuring that power consumption can both be optimized towards and reliably predicted scales from consumer applications, where batteries compete for space with other components in mobile devices, to enterprise applications, where the cumulative energy efficiency of many such systems incurs both economic and environmental costs. As a result, attention is increasingly moving from CISC to RISC designs, which, by nature of their design, require less transistors per die, and thus have a tendency[1] to consume less energy. The power requirements of embedded systems also play a deciding factor in deployments that must operate remotely or with constrained resources. Because the RISC-V ISA is an open standard, it can be scaled and adapted across this spectrum of embedded low-power use cases, ranging from medical implants to satellites [MR+23; Fur+22].

While many modern CISC processors provide hardware features to approximate power consumption such as Intel's RAPL [Dav+10], most RISC-based processors, especially those geared towards embedded applications, are limited in these feature sets, instead relying on software to extend functionality. Most current and previous research, such as that of Georgiou et al. and Lee et al., has focused on the older and more prevalent ARM architectures, with less practical tests being performed on the emerging RISC-V-based platforms [Geo+21; Lee+01].

This thesis will collect and analyze energy readings of the *ESP32-C3* series system on a chip (SoC), at the heart of which lies a modern RISC-V central processing unit (CPU). A configurable bare-metal benchmarking framework is developed around a set of representative workloads to collect hardware performance monitoring counter (PMC) values in a controlled fashion. The aggregated PMC data is then correlated with the system's current draw during the execution of said workloads, allowing for power modelling from perspective of the embedded system.

As embedded systems frequently have restricted hardware counters, an effort will be made to correlate the energy model with events in Zephyr, a contemporary real-time operating system (RTOS) that targets a multitude of platforms, extending the energy model via software tracing.

[1] It should be noted that actual power consumption varies strongly by implementation, and the above is a general trend. The exception proves the rule; x86-based processors do exist for embedded applications, and RISC ISAs have been developed for HPC [NKK04; Lee+23].

2 FUNDAMENTALS

This chapter introduces the fundamental general concepts and methodologies used in measuring the power consumed by an embedded processor. Section 2.1 begins with a principal explanation of the mechanisms involved in determining the power consumption of an electrical system. The equipment and fundamental software used to implement these concepts is then described in Sections 2.2 and 2.3. The techniques for quantifying event counts through hardware PMCs are elaborated upon in Section 2.4, providing a non-intrusive way to later correlate specific system activities with power consumption trends.

2.1 Measuring Power

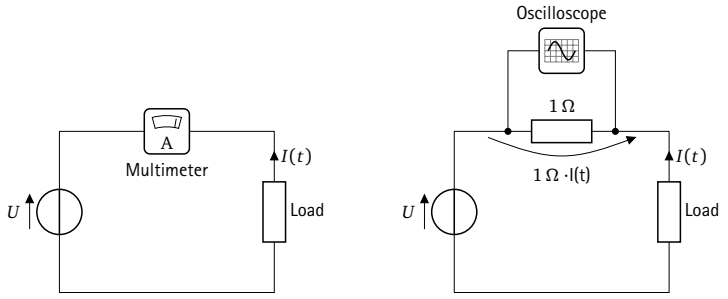
Electrical power, measured in watts (W), is defined as the product of electric potential, measured in volts (V) and denoted U, and current, measured in amperes (A) and denoted I. Given, for example, a constant DC voltage supply U, we can thus easily determine the power P of a variable resistive load (e.g. a microprocessor) at a point in time t in regards to the current at t as a function expressed as

$$P(t) = U \cdot I(t). \tag{2.1}$$

Figure 2.1a shows the textbook implementation of a current measuring setup, in which a commonly available multimeter with a low internal resistance is connected in series to measure the current drawn by a load at constant voltage [SP24]. Because the benchmarking processes in the later parts of this thesis can start and stop at very short intervals, an oscilloscope is substituted for a multimeter. These devices feature a much higher temporal resolution with the restriction that probes are usually limited to measuring only voltages. As a result, a high-precision shunt resistor must be used to measure voltage drop-off by exploiting Ohm's law in Equation (2.2) [SP24]. When using a resistor value of $R = 1\,\Omega$, this results in a one-to-one numerical conversion of voltage to current, the implementation of which is shown in Figure 2.1b.

$$U = R \cdot I \iff I = \frac{U}{R} \tag{2.2}$$

The oscilloscope captures n samples $(c_0, c_1, c_2, \ldots, c_n)$ in a given timebase $t = [0, s]$, called a *trace*. These samples vary from one another as the load's current draw changes over time. The arithmetic mean $(\bar{I})$ of the currents consumed serves as a practical reference value of the current over a period of time, and can be approximated from the trace through Equation (2.3). Because the shunt's resistance is simply a reciprocal linear factor of the mean, error tolerances of



(a) Basic circuit to measure current draw from a load.

(b) Circuit to measure current draw using a 1 Ω shunt.

Figure 2.1 – Near-equivalent current measurement methods.

the resistor can easily be compensated for by adding a coefficient. For example, a shunt which measures 1.10 Ω in actuality could be easily by compensated for by multiplying the mean by $(1.1)^{-1} = 0.\overline{90}$.

$$\bar{I} = \frac{1}{s}\int_0^s I(t)\,\mathrm{d}t \approx \frac{1}{n}\left(\sum_{i=0}^{n}\frac{c_i}{1\,\Omega}\right). \tag{2.3}$$

Assuming the approximate relationship between the variable load's state and current is bijective, it should be possible to predict future current draw by counting observable events alongside the load's current draw in controlled conditions.

2.2 Employed Hardware

The *ESP32-C3-DevKitM-1* development board was chosen as the embedded platform for practical measurements. The primary IC on board is the *ESP32-C3-MINI-1,* a RISC-V based SoC. This platform was chosen due to its ubiquity and implementation of custom control and status registers (CSRs) [Esp24c, pp. 29–38, 742]. The CPU is based upon the `rv32imc` microarchitecture with the `ilp32` application binary interface (ABI). It features the base 32 bit integer (`i`) ISA with multiplication/division (`m`) and compressed instruction (`c`) extensions. The significance of these extensions will be discussed in greater detail in Section 2.3. The development board also features pre-soldered pin headers, thereby enabling access to the SoC's onboard JTAG, a 5 V to 3.30 V LDO, as well as general-purpose input/output (GPIO) [Esp24a]. The specific uses of the aforementioned components are explained in Chapter 3.

2.2.1 Power Supply & Measurement

A *KORAD KD3005D* laboratory power supply unit (PSU) is used to provide a constant operating voltage to power the ESP32-C3-DevKitM-1. Because the PSU stabilized, jumps in current drawn by the development board have a negligible ($\leq 0.01\,\%$) effect on the output voltage [Don, p. 6].

The multimeter used to determine the initial baseline power consumption values as described in Section 3.1 was the *RIGOL DM3058E* digital multimeter. As the multimeter's sampling rate was too slow to effectively synchronize its sampling with the beginning and endings of the ESP32-C3's benchmarks, a *Rohde & Schwarz HMO3004* digital oscilloscope was used. Both

instruments support the USBTMC standard, which allows for the configuration, control and data acquisition from a PC programmatically.

2.2.2 Host PC

The host PC (from this point on referenced simply as "host") in use is a standard x86_64 workstation running Debian GNU/Linux 12. This PC cross-compiles benchmarks, controls the SoC, collects its benchmark data as well as the instruments' samples, and later uses this collected data to train predictive models.

2.3 Employed Software

In order to measure PMCs reliably, the SoC should run its benchmarks with minimal side-effects caused by interrupts or system calls. For this reason, the decision was made to program the processor *bare-metal,* without an operating system (OS). While possible, this method is atypical for the ESP32-C3, which usually boots into a FreeRTOS-based application image via a multi-stage bootloader [Esp24c, p. 192] using the Espressif IoT Development Framework (ESP-IDF).

In order to compile and execute C code to be executed without a full OS, an existing but apparently no longer actively developed software development kit (SDK) project titled *MDK* was hard-forked to *c3dk* [Lyu22; Arn24]. The forked project, as implied by its name, exclusively focuses on the ESP32-C3 and has been extended to support reading and writing to the SoC's RISC-V CSRs as well as the built-in USB-JTAG bridge. Most of both projects' functions were implemented "from scratch" (without FreeRTOS or ESP-IDF components), using the *ESP32-C3 Technical Reference Manual* and the published read-only memory (ROM) source code[2] published by Espressif. Important components include:

link.ld a link script which defines memory regions for the stack, mapping instruction RAM (IRAM) and data RAM (DRAM), as seen in Figure 2.2. This also allows access to module/peripheral registers[3] as listed in *ESP32-C3 Technical Reference Manual*, Table 3-3. The entry point of the program is defined to a function in the startup code;

boot.c entry point for all programs. This startup code initializes the heap, sets the processor clock frequency, and calls `main()`;

c3dk.h a header file which defines a multitude of useful preprocessor macros and statically inlined functions which can be called from other code. Most importantly, it provides a convenient interface to access registers as well as GPIO and JTAG pins;

Build System consisting of a `Dockerfile` which defines a containerized build environment, including the necessary toolchain, and a `build.mk` makefile stub, which can be included in projects and defines compilation rules using the container and bind mounts.

The ESP32-C3 contains a small subset of GNU compiler collection (GCC) standard library functions embedded in its ROM, which were initially linked against by the SDKs. During the porting process of benchmarks described in Section 2.5, it was quickly discovered that many other compiler routines required for extended arithmetic operations and floating point emulation (e.g. `__muldf3`) were not implemented, and as a result, had to be included separately by the host.

[2] `https://github.com/espressif/esp-idf/tree/master/components/esp_rom/esp32c3`

[3] Note that RISC-V CSRs can only be accessed via specialized instructions, as shown in Listing 3.1

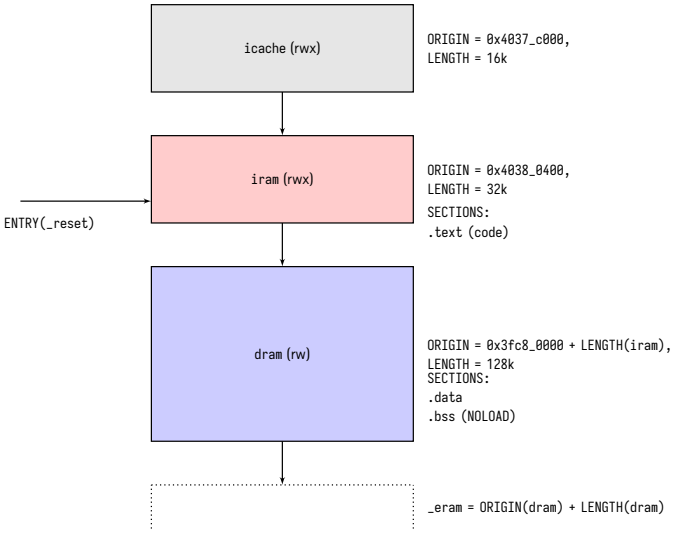


Figure 2.2 – Memory regions defined by the linker script.

Because the specific microarchitecture and ABI combination of the ESP32-C3 is not packaged by most distributions of GCC, a custom toolchain was configured and built from scratch using the source code[4] provided by the RISC-V International nonprofit organization. This tailored multilib toolchain is enables *complete* soft-float and GNU C-Library support. To avoid the repeatedly long build times involved with compiling GCC after executing `make clean`, the Dockerfile was adapted to use a multi-stage build process, allowing the container used by the SDK to be discarded while preserving the toolchain binaries compiled in the intermediate build container.

The *c3dk* project serves as the foundation for all bare-metal benchmarks later described in Chapter 3.

2.4 Methods and Techniques

Many microprocessors implement special registers that are used for event and time tracking, commonly referred to as PMCs, termed CSRs when specially implemented in RISC-V processors. The specific purposes and total number of events counted in these PMCs varies by microarchitecture, which in turn is designed to fulfill certain application goals (i.e. the balance between cost, performance and power consumption). For example, a very simple microcontroller may only be able to count cycles, while more advanced processors may be able to count memory accesses, as listed in Table 2.1. These registers are not only useful for research or debugging purposes, but are also used in RTOS scheduling and general performance analysis [XLT24; And15] and provide direct insights into how the human-readable source code of a program is actually executed on bare metal.

PMCs can be broadly generalized into two categories. A *fixed* counter is one that (if enabled) continuously tracks a single event, such as a clock source, and is commonly implemented as a timer on platforms that feature it. On the other hand, *programmable* counters can be configured

[4] `https://github.com/riscv-collab/riscv-gnu-toolchain`

μ-Architecture Example	RISC-V RV32IMC_Zicsr ESP32-C3	ARMv6-M Cortex-M0+ RP2040	ARMv8-M Cortex-M33 STM32-H5
Systick register size	54 bits	24 bits	24 bits
Total countable events	11	1	8
... Instruction count?	✓		✓
... Cycle count?	✓	✓	✓
... Load/store count?	✓		✓

Table 2.1 – Comparison of hardware counters between three ISAs typically used in embedded systems.

on-the-fly using special instructions or other registers, such as ARM's Data Watchpoint Trace (DWT) [ARM24, p. 77].

2.4.1 Determining Event Count

32 bit RISC-V, like other CPU architectures, incorporates a variety of instructions which inherently differ in their effective time complexity and the hardware resources they engage. For example, a `LOAD` instruction which interfaces with (slower) main memory may require more time and power than an `ADDI` instruction, which interfaces with other (fast) registers and typically completes within one clock cycle [VO22].

As the sequential execution of instructions of an embedded processor can be viewed as largely deterministic[5], it can be expected that repeatedly executing the same instruction sequences will result in the same changes in PMC values and power consumption. These instructions, if used as a point of reference for comparison with other instruction sequences, can then be called *benchmarks*. Ideally, each benchmark varies in its relative event frequencies and power consumption to provide a broad range of values to form correlations with. For example, one benchmark may execute mainly integer operations, while another is intentionally memory-intensive, and as such should result in different time and power behavior when executed. Provided a large collection of samples and subsequent events are recorded, these can then be aggregated to determine how a specific correlates with power.

2.4.2 Modelling Power Correlations

After a benchmark completes, the p events counted by the processors PMCs can be read out and mathematically represented by a p-dimensional vector together with the mean current y. It is often difficult to compare benchmarks directly to one another, as the total number of events recorded may vary by orders of magnitude between them while retaining a smaller difference between mean current. As such, it is beneficial to determine the relative frequency of an event occurring in a benchmark by means of *unit normalization,* as represented in Equation (2.4), assuring that a comparison of the underlying patterns can take place regardless of magnitude.

$$\hat{\boldsymbol{x}} = \frac{1}{\sum_{i=1}^{p} e_i} \begin{bmatrix} e_1 & e_2 & \dots & e_p \end{bmatrix}^T = \begin{bmatrix} x_1 & x_2 & \dots & x_p \end{bmatrix}^T \tag{2.4}$$

[5] Jitter, random number generation (RNG), caching or speculative execution *will* result in slight differences, which can be accounted for by taking a large number of samples.

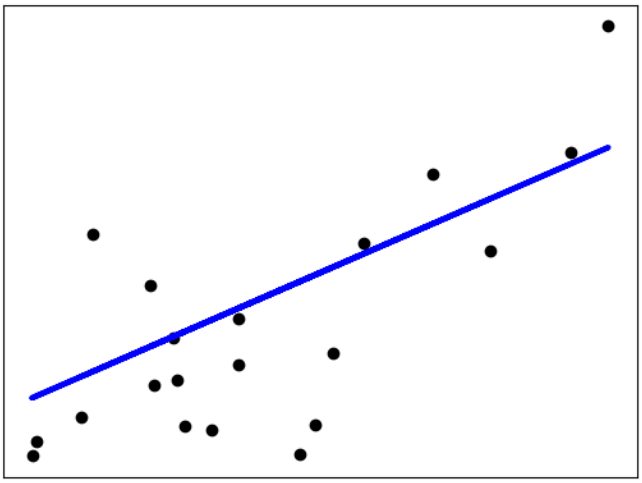


Figure 2.3 – Example of a two-dimensional ordinary least squares (OLS) regression by "Scikit-learn: Machine Learning in Python."

While the majority of instructions executed as part of a benchmarking algorithm modify the processor's state (either through registers or the pipeline), it can be assumed for the sake of simplicity that the relationship between an event's frequency and the mean current is linear: if[6] an event has a measurable impact on the current draw of the processor, the mean current draw of the benchmark as a whole will likely correlate by a real coefficient as the frequency of this event increases. As demonstrated by Figure 2.3, the resulting prediction (blue line) is meant to follow a trend in data as close as possible.

By collecting the resulting PMC states across different benchmarks a total number of n times, one can define a feature matrix $\boldsymbol{X}$ together with the vector of resulting mean currents, $\boldsymbol{y}$, as

$$\boldsymbol{X} = \begin{bmatrix} \hat{\boldsymbol{x}}_1 \\ \hat{\boldsymbol{x}}_2 \\ \vdots \\ \hat{\boldsymbol{x}}_n \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,p} \\ x_{2,1} & x_{2,2} & \dots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,p} \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Using a vector of unknown weights $\boldsymbol{w} = \mathbb{R}^p$, the mean current can now be solved for by a linear combination of each feature with a weight, as seen in Equation (2.5) [Ped+11]. While there are many possible approaches to determining the correct features and their associated weights, the perhaps most common one is now known as the ordinary least squares (OLS) regression, the discovery of which is attributed to Legendre in 1806, now an elementary machine learning (ML) algorithm.

$$\hat{y}(\boldsymbol{w}, \boldsymbol{x}) = w_0 + \sum_{i=1}^{p} (w_i f_i) = w_0 + w_1 f_1 + \ldots + w_p f_p \tag{2.5}$$

[6] It is, of course, possible for an event frequency to have *no* meaningful correlation if mean currents are too variate for the given frequency range across all benchmarks.

Note that w_0 is an independent constant not paired with any feature, and serves as the *bias,* also termed *intercept* [Ped+11].

Given the considerable number of variables and potential for error (noise) in the measured current, it is highly probable that the linear system will have no perfect solution. Instead, the OLS method seeks to identify the "best fit" for each weight component in $\boldsymbol{w}$ by finding the smallest possible squared value of the error for each weight, as seen in Equation (2.6), termed the *objective function* [Ped+11].

$$\min_{\boldsymbol{w}} ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 \tag{2.6}$$

2.5 Related Work

In the past, work done by Walker et al. was able to determine a relationship between PMC events and their impact on power consumption on ARM Cortex-A7 and Cortex-A15 CPUs. Methods were employed to avoid multicollinearity between weights in a linear model, which resulted in an accuracy of 3.80 % and 2.80 % average error, respectively [Wal+17]. Rodrigues et al. were able to predict energy consumption with 95% accuracy using just three counters on Intel x86_64 architectures (# fetched instructions, L1 cache hits, and dispatch stalls) using 38 benchmarks. The authors demonstrated that the power metric from a high-performance core could be employed to estimate power on a low-performance core of the same ISA [Rod+13]. The transfer of a power model from one platform to another was also attempted by Nikov et al., which employed a dual-platform approach using an field programmable gate array (FPGA) to model power for a space-rated LEON3 processor which does not implement PMCs. The baseline power consumption from Dengler et al. is used as a comparison point for the baseline power consumption and further discusses energy-aware scheduling [Den+23].

Mair et al. lay some foundational best practices in "Myths in PMC-Based Power Estimation." While not all points discussed apply to the small, passively cooled RISC development board used in this thesis, their observation on memory-related PMC events also applies to the later analysis in Chapter 4[Mai+13].

In the perhaps most influential work for this thesis, Pallister, Hollis, and Bennett created a pool of benchmarks for embedded systems to measure energy consumption based on integer and floating point operations, memory access intensity, and branching frequency in their paper "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms." The algorithm executed in each benchmark was specifically chosen to simulate common workloads of embedded systems across five broad categories, namely for *security, network, telecommunications, automotive,* and *consumer* applications. The authors selected ten benchmarks which appeared to be the most suitable for power modelling, which will be adapted for the ESP32-C3 in this thesis [PHB13].

It is also worth mentioning that the RISC-V target of this thesis, the ESP32-C3, has a hardware abstraction layer (HAL)[7] written in the Rust programming language, and there exists at least one PMC related benchmarking project[8] using it.

[7] https://docs.esp-rs.org/esp-hal/esp-hal/0.20.1/esp32c3/esp_hal/index.html

[8] https://github.com/onsdagens/esp32c3-rt-benchmarks

3 ARCHITECTURE

The rationale and physical setup of the experiment to record current and PMC data are outlined in this chapter. Section 3.1 begins by describing the hardware connections between the PSU, the development board, various measurement instruments, and the host alongside initial baseline current values. Section 3.2 then describes how PMCs are accessed by the benchmarking firmware image running on the SoC, after which Section 3.3 illustrates the complete benchmarking sequence and software components required to coordinate and execute benchmarks.

3.1 Baseline Configuration & Values

The ESP32-C3-DevKitM-1 features three ways to provide (and thus measure) power:

- 5 V DC, via the LDO, provided via the host's USB
- 5 V DC, via the LDO, provided via the `5V` and `GND` pins
- 3.30 V DC, provided directly via the `3V3` and `GND` pins

Because a reliable physical serial connection between the host and the development board is required for flashing, erasing, and transferring data, it is critical to ensure that the connection will not interfere with the measured current in one of two ways: Voltage from the host's 5 V USB should not reach the development board, interfering with the stabilized reference voltage provided by the external power supply, and the USB connection must not cause additional components (e.g. the USB-UART of the development board) to sporadically activate, resulting in increased power consumption.

The ESP32-C3 SoC provides an unpowered serial interface via its integrated USB Serial/JTAG Controller which is independent of the USB-UART of the development board and can be accessed via the GPIO pin headers. To utilize this interface, an off-the-shelf shielded USB-A cable was stripped to expose the white and green differential data wires (`D-` & `D+` respectively) along with the `GND` wire. These individual cables were then stripped and crimped with *Mini-PV* receptacle connectors[9] to provide a secure connection to the pins on the development board. In this configuration, both the multimeter as well as the oscilloscope can be used to measure current (see Figure 2.1 in Chapter 2). The final configuration used in Section 3.3 to collect benchmark data is illustrated in Figure 3.1. As the oscilloscope is connected to the host's ground via USB

[9] Also known under the genericized trademark "DuPont connector"

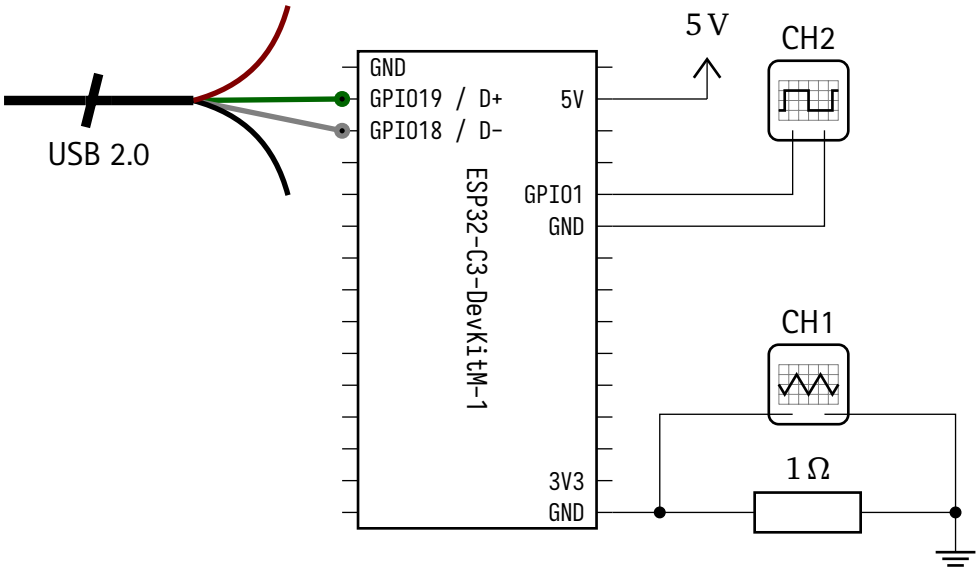


Figure 3.1 – Powering the development board with an external 5 V supply. Note that `CH1` and `CH2` are part of the same oscilloscope, and thus share the GND connection.

and its ground probe was attached to the ESP32-C3-DevKitM-1's `GND`, the need for a separate USB ground between the host and development board is annulled and could lead to a short circuit.

To provide a general idea of the current draw of an idling processor, baseline values are needed. To determine baseline power consumption in different configurations, the 4 MiByte onboard SPI flash chip was fully erased or flashed with an image containing a minimal firmware. The multimeter was then set to the *slow* current sampling mode, allowing for $5\frac{1}{2}$ digit sampling precision.

After booting the chip, a Python script collected samples from the multimeter while the processor was idling in each configuration. Each sample read via the USBTMC interface was parsed to a `decimal.Decimal` data type to ensure good floating point accuracy, with the arithmetic mean ($\bar{I}$) and standard deviation (σ) being calculated after recording 1000 samples. The resulting values for each voltage and CPU frequency (f) are shown in Table 3.1.

Configuration			CPU	Current	
Pin	Flash Contents	Connections	f	$\bar{I}$	σ
3V3	erased	—	20 MHz	8.68 mA	8.14 µA
5V	erased	—	20 MHz	9.04 mA	6.2 µA
5V	erased	JTAG	20 MHz	9.25 mA	82.91 µA
3V3	minimal image	—	160 MHz	27.76 mA	7.05 µA
5V	minimal image	—	160 MHz	28.8 mA	19.23 µA
5V	minimal image	JTAG	160 MHz	28.83 mA	14.77 µA

Table 3.1 – Baseline currents at different voltages. No serial data was transferred during the evaluation.

With an empty flash chip, the ESP32-C3 boots with the default CPU clock frequency of 20 MHz [Esp24c, Section 6.2.3] and starts a watchdog timer which resets the chip after 100 ns [Esp24c, Section 12.2.2.2] of inactivity. The standard deviations for most configurations are well within the DC current noise tolerances of the multimeter [RIG24, p. 2], with the exception of the JTAG connection. These larger deviations in power are due to the ROM bootloader printing information after the watchdog triggers a reset, which can only be circumvented by loading a firmware image and disabling the watchdog to allow for true idling.

The minimal firmware image, built upon *c3dk*, contains a small setup routine in the boot code which sets the CPU frequency source to the 320 MHz phase-locked loop (PLL) source with a divisor of 2, resulting in a stable 160 MHz clock, and a `main()` function which disables the watchdog timer [Arn24]. The processor is then left in an idling state where current draw can be measured. The negligible difference in current draw caused by attaching the JTAG in the 5 V configuration makes this configuration suitable for measuring power while collecting data. The circa 1.04 mA increase in current when powering the board with 5 V instead of 3.30 V can be attributed to the onboard LDO voltage regular and power LED, which are conditionally activated when the board is supplied with 5 V. As the USB-JTAG connection can only be established with the 5 V power configuration and the bare-metal benchmarks run at 160 MHz, 28.83 mA will be used as a comparison value. This current aligns with the typical current consumption to be expected according to the manufacturer's datasheet [Esp24b, p. 21]. Thus, we can define an appropriate idle power consumption as

$$P_{\text{idle}} \approx 95.13\ \text{mW} = 3.30\,\text{V} \cdot 28.83\,\text{mA}. \tag{3.1}$$

This idling wattage is just under that described by Dengler et al. [Den+23, p. 7]. Note that, despite providing an external input power of 5 V, the ESP32-C3 chip itself is still powered by 3.30 V.

3.2 Reading ESP32-C3 CPU Events

The ESP32-C3 implements the "Zicsr" extension for accessing and modifying CSRs. In place of further implementing the standard "Zicntr" and "Zihpm" extensions for counters and timers [Wat+19, p. 50], the processor instead implements a single 32 bit custom machine performance counter, configured by three registers in the address space reserved by the RISC-V standard for custom use. These custom PMCs can be accessed by RISC-V-specific CSR assembly instructions as seen in Listing 3.1, and have been implemented in *c3dk*'s header file, allowing a benchmarking framework to access and modify the registers in a C program. These macros behave like regular functions, with the condition that the CSR address must be defined at compile-time.

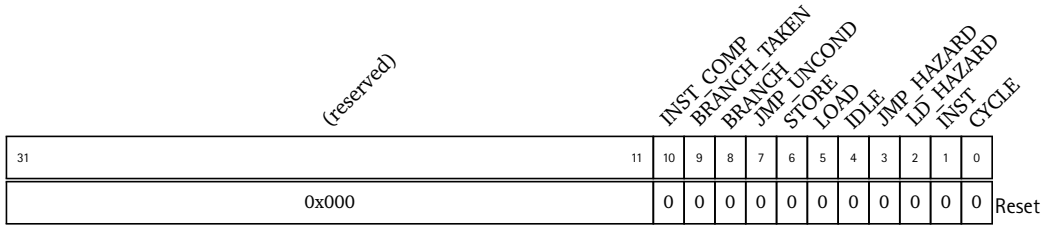
```
1  // Return value from CSR
2  #define csr_read(addr) __extension__ ({ \
3      uint32_t __tmp; \
4      asm volatile ("csrr %0, " #addr : "=r"(__tmp)); \
5      __tmp; \
6  })
7
8  // Write value to CSR
9  #define csr_write(addr, value) __extension__ ({ \
10     asm volatile ("csrw " #addr ", %0" :: "r"(value)); \
11 })
```

Listing 3.1 – GCC C preprocessor macros to read and write CSRs.

Two additional 54 bit hardware system timer are also available [Esp24c, pp. 30, 262], configurable to set alarm-style interrupts[10]. These operate using the 40 MHz Xtal clock source, passed in alternating ticks to a $\frac{1}{3}$ and $\frac{1}{2}$ fractional scaler, effectively resulting in a $\frac{40\,\text{MHz}}{2.5} = 16\,\text{MHz} = 62.50\,\text{ns}$ timer resolution [Esp24c, p. 263]. This specific resolution proves to be useful in benchmarking as it is an independently sourced 10 : 1 scale of the frequency set by *c3dk* during startup as well as a high-resolution wall-clock. The elapsed clock ticks are read from memory-mapped high (22 bit long) and low (32 bit long) registers, *not* specialized CSRs.

Despite being *de facto* limited to only timers and a single programmable general purpose counter, a wider set of events can be measured compared to similar embedded processors, as shown in Table 2.1.

Register 3.1 – mpcer (0x7E0)



INST_COMP Count Compressed Instructions. (R/W)

BRANCH_TAKEN Count Branches Taken. (R/W)

BRANCH Count Branches. (R/W)

JMP_UNCOND Count Unconditional Jumps. (R/W)

STORE Count Stores. (R/W)

LOAD Count Loads. (R/W)

IDLE Count IDLE Cycles. (R/W)

JMP_HAZARD Count Jump Hazards. (R/W)

LD_HAZARD Count Load Hazards. (R/W)

INST Count Instructions. (R/W)

CYCLE Count Clock Cycles. Cycle count does not increment during wait-for-interrupt mode. (R/W)

The programmable PMC at offset `0x7E2`, designated machine performance counter count register (`mpccr`), is a full 32 bit register representing the current event count value. As this register is read/write, the counter can be reset to 0 or otherwise adjusted, if needed. Its behavior is configured by the `mpcer` and `mpcmr` registers at offsets `0x7E0` and `0x7E1`, respectively.

[10] Purposefully not enabled in the context of benchmarking.

The mpcmr register contains two bit flags to control the event counting behavior. COUNT_EN enables event counting *per se,* while COUNT_SAT either halts the processor upon reaching the maximum mpccr value (=1) or allows the counter to overflow (=0) [Esp24c, p. 36]. By default, these bits are both set to 1, which ensures that selected events are counted and mpccr will not overflow after $2^{32}-1$ events[11], the desired behaviour when collecting benchmarking data.

The bit fields of the mpcer register as described in the *ESP32-C3 Technical Reference Manual* are shown in register diagram 3.1, which enables the specific events to include in the total event count. Although it is possible to enable multiple bits in the field, it is noted that "each bit selects a specific event for counter to increment. If more than one event is selected and occurs simultaneously, then counter increments by one only." [Esp24c, p. 36] Because the ESP32-C3's CPU incorporates a four-stage pipeline, it is extremely likely that events will occur simultaneously, and therefore it is not possible to guarantee an accurate count of simultaneous events. As a result, each event of the 11 bit wide mpcer register must be counted independently by repeating benchmarks. Nonetheless, 100 samples were collected for *both one and two* possible event combinations as an extra precaution, resulting in $100 \cdot \left[\binom{11}{1} + \binom{11}{2}\right] \cdot 10 = 66000$ individual samples for all of the ten BEEBS benchmarks used as the foundation for the final experiment.

In a straight-line code sequence without branching or calls, individual pipeline stages allow all subcomponents of the CPU to be utilized simultaneously. For example, as one instruction is being executed, the successive instruction is already being decoded concurrently. This, however, opens up the possibility of an instruction in one stage of the pipeline interfering an instruction in another stage, resulting in data or control dependencies which must be resolved by *bubbling* the pipeline with no-operations (NOPs), and are counted by mpcer as LD_HAZARD and JMP_HAZARD events, respectively.

3.3 Building and Executing Benchmarks

To build and execute code without an RTOS, it was decided to fork and and extend the *MDK* project to create a small, modern SDK focusing on the ESP32-C3. More information on the work done on this foundational piece of software can be found in Section 2.3.

The host PC must also manage the configuration and compilation of benchmarks, control the development board (described in 3.3.2), interface with the oscilloscope via USBTMC, and manage resulting data. Due to the large scope of functionality, various Python modules (files) were bundled into a package named *autobench.* The root of this package contains an executable __main__.py file, which uses the *versuchung* package[12] to handle inputs (e.g. a directory containing benchmarks) and benchmarking parameters. When executing autobench, the host iterates through the combination of all specified benchmarks and the selected mpcer bitmasks specified in Section 3.2, and then executes the steps described in the following Sections 3.3.1, 3.3.3 and 3.3.4.

3.3.1 Preparing for a Benchmark

Before the benchmarking process begins, the oscilloscope must be set-up via specific USBTMC commands and the benchmark itself must be built and flashed to the SoC, the detailed sequence of events being visualized in Figure 3.2.

[11] It would take 26.84 s to overflow when counting clock cycles at 160 MHz.

[12] https://pypi.org/project/versuchung/

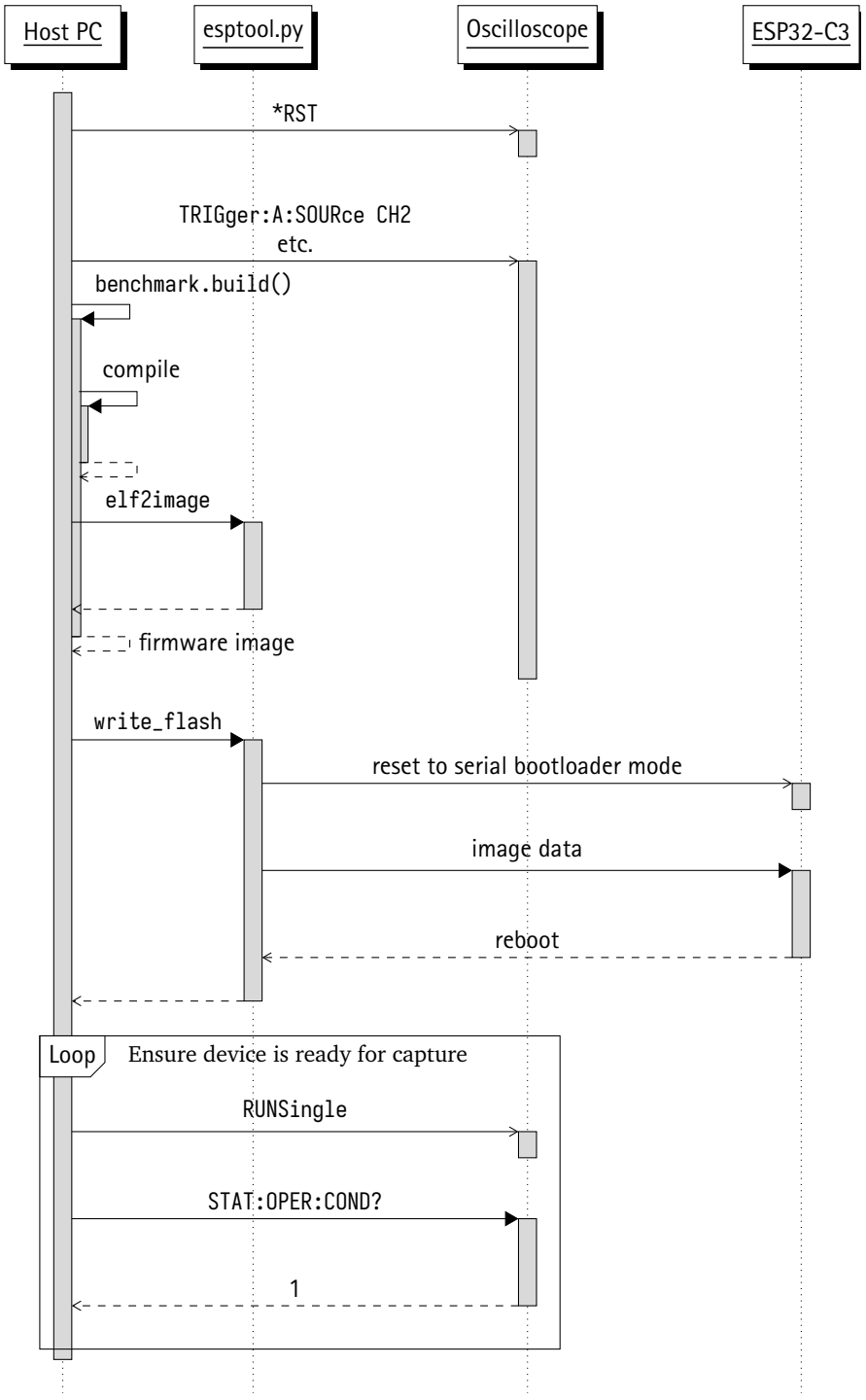


Figure 3.2 – Setup sequence in preparation for a benchmark. As the Oscilloscope sets up after a reset, a bootable firmware image is built and flashed. Before a benchmark is initiated, the host PC waits an operational status register bit to be set, indicating that all commands have been processed and the device is ready to capture.

To ensure a known instrument state, the oscilloscope is reset (`*RST`) and configured with a series of commands which set voltage scales and an initial timebase of 100 ns. One channel (`CH1`) is configured to monitor the voltage drop across the shunt resistor with the scale 8 mV, and the other (`CH2`) is set to trigger when the development board's benchmark state signal flanks *high*[13], indicating a measurable benchmark has begun.

Each specified benchmark is then compiled using to an ELF binary by autobench, using the *espbench* wrapper, which implements the serial control protocol and benchmarking setup described in Sections 3.3.2 and 3.3.3. This binary is then passed to Espressif's *esptool*, which contains the `elf2image` and `write_flash` subcommands to convert the executable binary into a bootable image for the SoC and subsequently writes it to the flash chip. The SoC is then rebooted and the oscilloscope is set capture a single trace by issuing the `RUNSingle` command. The operation status condition register of the instrument is queried until it reports the device is ready for capture.

3.3.2 Serial Benchmarking Control Protocol

In order for the host to effectively coordinate between instruments and control actions of the SoC, a basic control protocol had to be implemented. Upon boot, the SoC begins waiting for one of the command bytes listed in Table 3.2 and executes the desired action, after which it returns to listening again.

PC Command Byte	ESP Response Byte(s)	Purpose
`'p'`	`'p'`	**P**ing to determine connectivity
`'b'`	`'o'`	Begin **b**enchmark, response: **o**kay
`'r'`	*binary data*	Request **r**esult of benchmarks

Table 3.2 – Serial command protocol showing request and expected response

The first and simplest command (`'p'`) is implemented as a simple ping-style echo. This is used to ensure that the SoC has its JTAG interface enabled and is ready to accept the next command. The second command (`'b'`) tells the SoC to begin the benchmarking procedure. This command is necessary in order to ensure that the benchmark only begins after the oscilloscope has been set-up and configured. The response is sent as a short confirmation to that the benchmark will start, during which the JTAG is disabled.

The final command (`'r'`) requests the data after the execution of a benchmark, the process of which is further described in 3.3.4.

3.3.3 Benchmarking Procedure

Upon receiving the command byte to begin a benchmark, the SoC disables the JTAG by setting the bit `USB_SERIAL_JTAG_USB_PAD_ENABLE` (number 14) of the register `USB_SERIAL_JTAG_CONF0_REG` (offset `0x018`), as during initial benchmarking trials with the oscilloscope, it was found that simply opening the serial device file resulted in large current fluctuations as seen in Figure 3.4a. Immediately after disabling the JTAG, current draw drops to baseline levels and the event sequence shown Figure 3.3 proceeds between the three devices.

[13] High-level output is defined as a voltage $\geq 0.8 \cdot 3.30\,\mathrm{V}$ [Esp24b, p. 20].

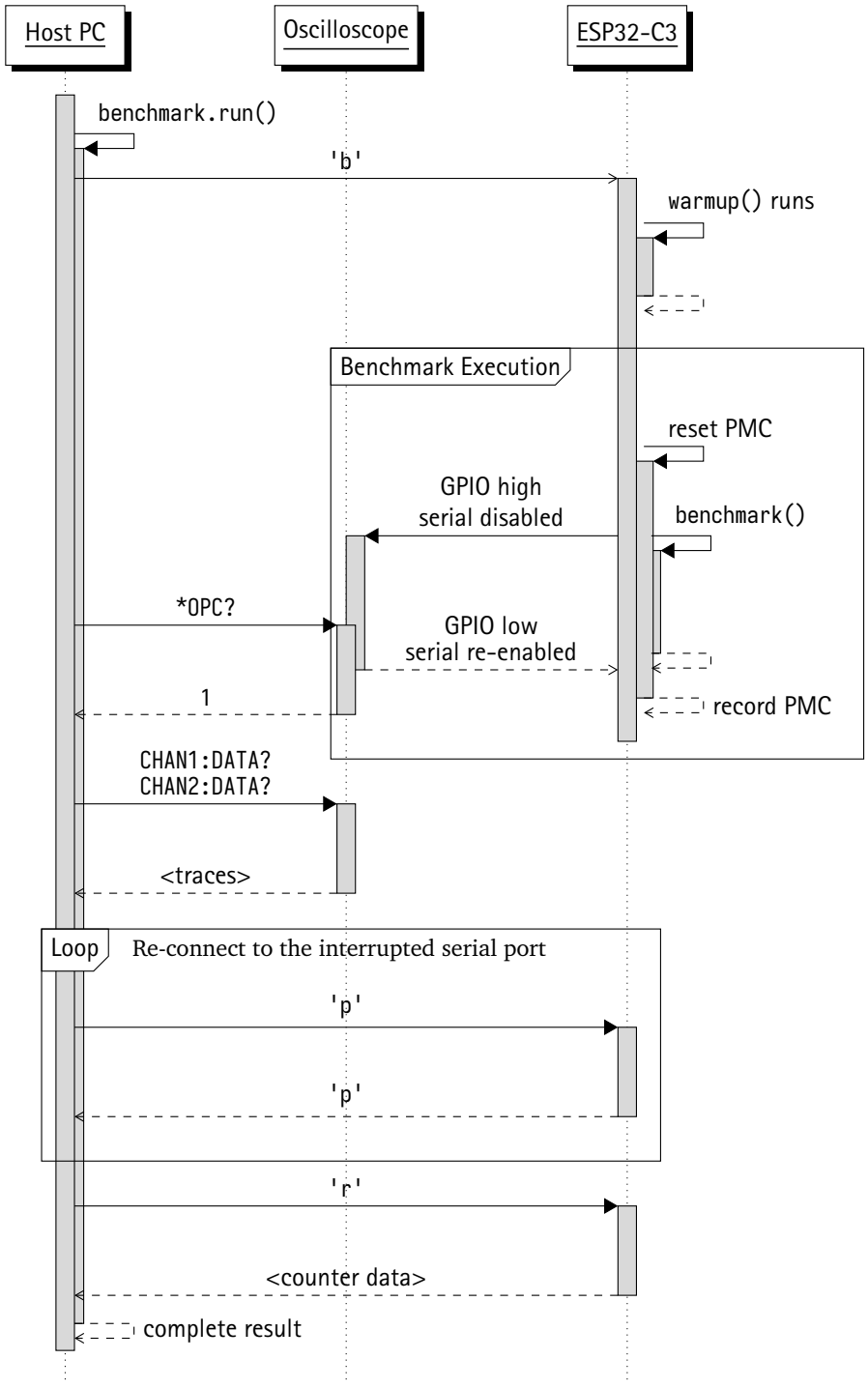
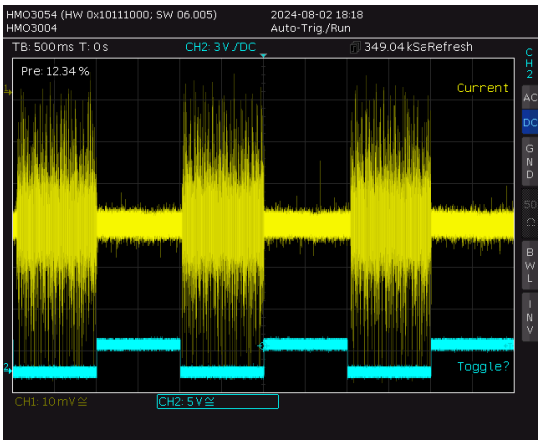
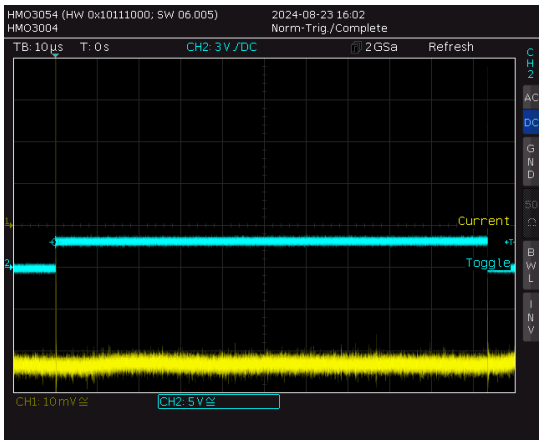


Figure 3.3 – Benchmark execution sequence. Writing `'b'` to the serial line triggers the ESP32-C3 to *begin* preparation for a benchmark, while `'r'` triggers the ESP32-C3 to write benchmark *results* to the serial port. The SCPI command `*OPC?` blocks until the full trace has been captured after a trigger event.

The processor now begins a warm-up phase, in which the `initialise_benchmark()`[14] and `benchmark()` functions are repeatedly[15] called. Directly before calling the benchmarking function for the actual measurement, the system tick count and `mpcer` values are stored for later verification on the host, the benchmarking signal pin is set to high to trigger the oscilloscope, and the `mpccr` register is reset to 0.



(a) Demonstration of the current interference caused by toggling the JTAG on (when the indicator signal (`CH2`) is set to low)



(b) Single trace of a benchmark from start to finish. The current stays at a continuous level, even outside of the indicator signal (`CH2`) due to warm-up and -down rounds.

Figure 3.4 – Captured oscilloscope traces, with `CH1` showing voltage drop-off

As soon as the benchmarking function returns, the `mpccr` and system tick timer contents are read and the benchmarking signal pin is set to *low*. Warm-down rounds are run right before re-activating the JTAG to avoid the risk of contaminating oscilloscope's trace with a current spike directly at the end of a benchmark, demonstrated in Figure 3.4b.

3.3.4 Collecting, Decoding and Saving Results

Benchmark data is collected on-board the SoC using the C data structure as shown in Listing 3.2, and output directly to serial when the `'r'` request byte is processed.

```
1 typedef struct Result {
2   uint64_t bitmask;
3   uint64_t start;
4   uint64_t end;
5   uint64_t pmc;
6 } Result;
```

Listing 3.2 – The resulting data passed to the host after a benchmark.

Because the data structure uses the same byte-aligned type for all fields, it is trivial to interpret and unpack the raw data on the host using the Python standard library's built-in `struct`

[14] This function is a part of all BEEBS benchmarks and serves to set up any needed state, such as initializing variables or seeding the random number generator.

[15] 100 times by default, overridable with a compile-time flag.

module[16] with the format string `"<QQQQ"`, denoting four little-endian unsigned long long (64 bit) integers. As the length of 4×8 Byte is known to the host, a re-transmission can be requested if the read number of bytes does not equal the expected number.

The whole process on the host side is blocked until the oscilloscope reports that a complete (`SINGLE`) trace has been captured, implying that the begin of a benchmark triggered the oscilloscope. The host then reads the oscilloscope's `CH2` trace. If the last value of `CH2` trace is high, it can be assumed that the timebase was too small to capture the entire duration of the benchmark's power consumption. The timebase is doubled, and the process is repeated until a trace is collected which was triggered by a high signal and ends in a low one, a case in which it can be assumed that the current was monitored during a full benchmark execution, after which `CH1` is queried and trimmed down to the period where `CH2` is high. From this series of data the minimum, maximum, mean, and median values are computed. These statistics, along with the fields described in Listing 3.2 and sample metadata such as benchmark name are saved to a Pandas `DataFrame` for later serialization and processing [tea24].

The architecture allows for capturing all of the bare-metal data needed to train a model. After successfully collecting the required samples from all benchmarks, the samples, each consisting of the benchmark's name (required for later aggregation operations) along with the CPU tick count, the `mpcer` and `mpccr` bitmasks and the mean current drawn are finally serialized to a Parquet file. This column-oriented file format was chosen over the more common comma-seperated values (CSV) file format because its self-describing nature leaves little room for misinterpretation of data types [Voh16].

[16] `https://docs.python.org/3/library/struct.html`

4 ANALYSIS & MODELLING

Understanding the relationships between the recorded PMC events and current draw is critical to effectively estimating power. Section 4.1 scrutinizes the aggregated samples collected from each of the BEEBS benchmarks collected in Chapter 3, which are then correlated with current consumption in Section 4.2. Building upon concepts introduced in Chapter 2, these observations are then used to evaluate different models, the predictions of which are then validated using PMC data from new, foreign benchmarks in Section 4.3. Finally, this analysis is extended to a real-time operating system in Section 4.4.

4.1 Benchmark Time and Power Metrics

The source code of each of the ten benchmarks chosen by Pallister, Hollis, and Bennett was compiled and executed with autobench. The mean power draw and CPU ticks elapsed for each benchmark are shown in Table 4.1. Because the mean value (μ) and standard deviation (σ) can vary strongly between each of the benchmarks, the percentual coefficient of variation (CV), defined in Equation 4.1, is deemed a useful metric to determine the distribution of sampled values between benchmarks. Because this value is low across the board ($< 2.50\,\%$), it can be inferred that the benchmarks were executed in a mostly deterministic fashion.

$$\mathrm{CV} = \frac{\sigma}{\mu} \times 100\,\% \tag{4.1}$$

Because the benchmarks cover a wide range of time and current draw (Figure 4.1), they appear to provide a suitable range of targets for a regression model. The only notable outlier is the forward discrete cosine transform (FDCT) benchmark, which in the BEEBS implementation, is almost exclusively based on a series of integer operations. As a result, this benchmark executes very quickly, needing around only 178 ticks on average (circa $11.12\,\mu$s) and a large amount of power (34.34 mA).

Despite its quick execution time, it shares superficial properties with the other two relatively high-power benchmarks, SHA256 and matmult-int (3$^{\mathrm{rd}}$ and 1$^{\mathrm{st}}$, respectively), both of which are also memory and especially integer-focused. Notably, FDCT performs a `memcpy()` call, which may have side effects in power draw. While the Blowfish and SHA256 benchmarks call this function as well, the extraordinarily short duration of the FDCT benchmark may still skew median and mean power draw samples as a result.

Another interesting metric are the cycles per event (CPE) shown in Figure 4.2, used to measure how many processor clock cycles are required, on average, to complete a specific

Benchmark	Power mean	Power σ	Power CV	CPU Ticks mean	CPU Ticks σ	CPU Ticks CV
blowfish	109.25 mW	208 μW	0.19 %	88 425.26	429.94	0.49 %
crc32	107.68 mW	877 μW	0.81 %	1732.08	41.86	2.42 %
cubic	105.19 mW	150 μW	0.14 %	99 850.62	8.72	0.01 %
dijkstra	107.82 mW	283 μW	0.26 %	68 334.39	168.14	0.25 %
fdct	113.32 mW	1.90 mW	1.68 %	177.27	0.66	0.37 %
fir2d	106.31 mW	534 μW	0.50 %	2691.06	2.66	0.10 %
matmult-float	107.74 mW	245 μW	0.23 %	20 097.50	4.14	0.02 %
matmult-int	113.59 mW	553 μW	0.49 %	8732.03	72.04	0.82 %
rijndael	108.09 mW	183 μW	0.17 %	81 444.01	10.39	0.01 %
sha256	110.24 mW	658 μW	0.60 %	890.73	1	0.11 %

Table 4.1 – General power and time statistics of the BEEBS benchmarks.

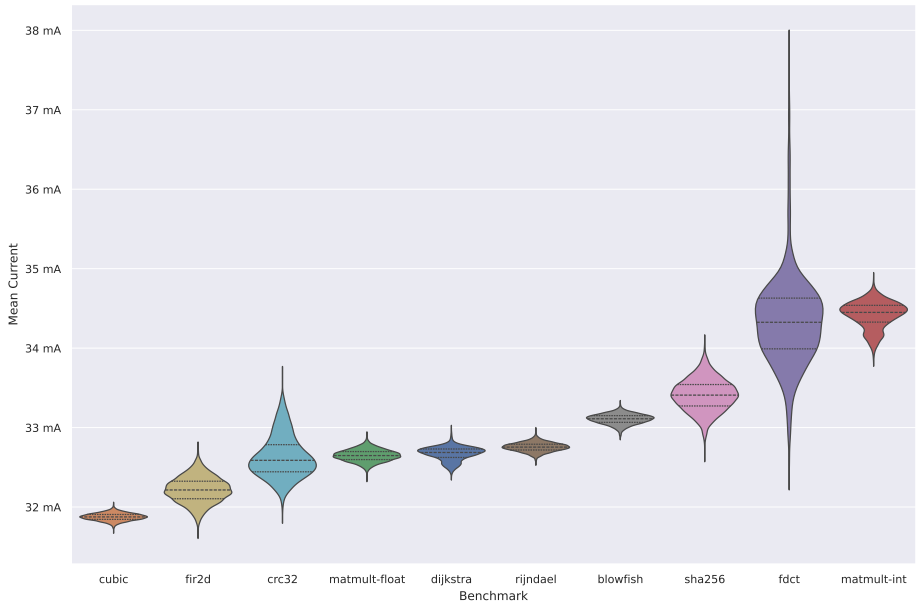


Figure 4.1 – Mean current consumption across 660 samples per benchmark, with quartiles marked for each.

event or task during the execution of a benchmark. This metric can easily be calculated by aggregating the mean number of events in a benchmark and dividing these by the mean sum of clock cycles needed to complete a benchmark. Section 3.2 notes that the timer register operates at 16 MHz, one tenth the speed of the independently sourced processor clock. As such, it is no surprise that the `CYCLE` and `INST` counts congregate around 0.10 ticks, aligning with the findings

of Van Overveldt [VO22]. Noteworthy is the pronounced distribution of BRANCH, BRANCH_TAKEN, JMP_UNCOND and LD_HAZARD events, which are distributed along multiple orders of magnitude, indicating strong differences in the low-level execution of benchmarks.

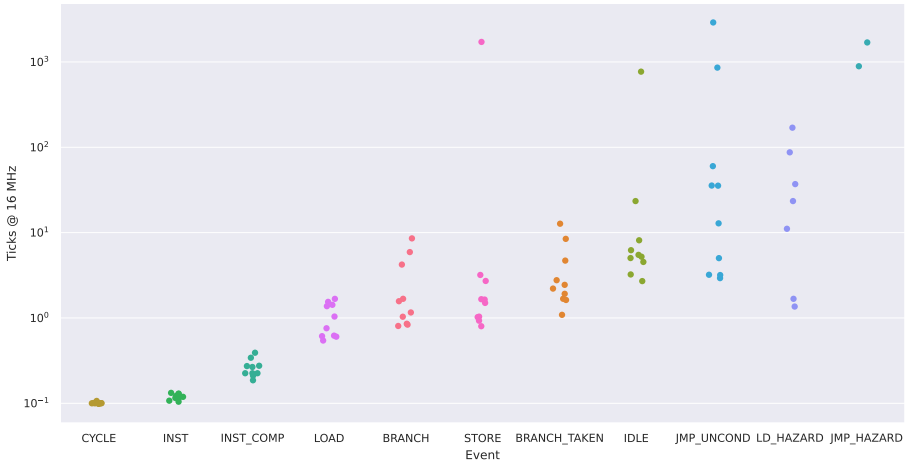


Figure 4.2 – Strip plot showing the CPE for each of the ten BEEBS benchmarks, where applicable.

4.2 Relationship between Events and Power Consumption

As noted at the end of Section 3.2, the ESP32-C3 is only capable of counting a single event type at a time. To ascertain the number of events occurring in the sampled benchmarks, the data can be aggregated into a pivot table, where the benchmark name serves as the row index and a column is allocated for each bit in `mpcer`. Aggregation operations can then be applied for each of the recorded samples. Table 4.2 shows the standard deviations of recorded event counts, which are extremely consistent. The only notable exception is *rijndael,* which uses RNG to yield a long sequence of random numbers which likely causes the large deviations compared to other benchmarks. Because events are consistent between samples, the mean between samples of a pivot table appears to be a suitable aggregate metric to estimate for the number of events per benchmark, as seen in Figure 4.3, which can be used to compensate for the ESP32-C3's lone event counter. An insignificant deviation also occurs for crc32 with the event bitmask 8 (`IDLE`), which can be attributed to load hazards.

Because events can vary greatly between benchmarks due to different execution times, it is difficult to directly compare the number of a particular event between benchmarks. Therefore, their relative frequency can be determined by normalization, as described by Equation (2.4) in Section 2.4.2.

The most power-intensive benchmarks in Figure 4.1, such as FDCT, SHA256 and matmult-int also present a relatively high (brighter) share of memory-intensive operations such as `LOAD` and `STORE` in Figure 4.3.

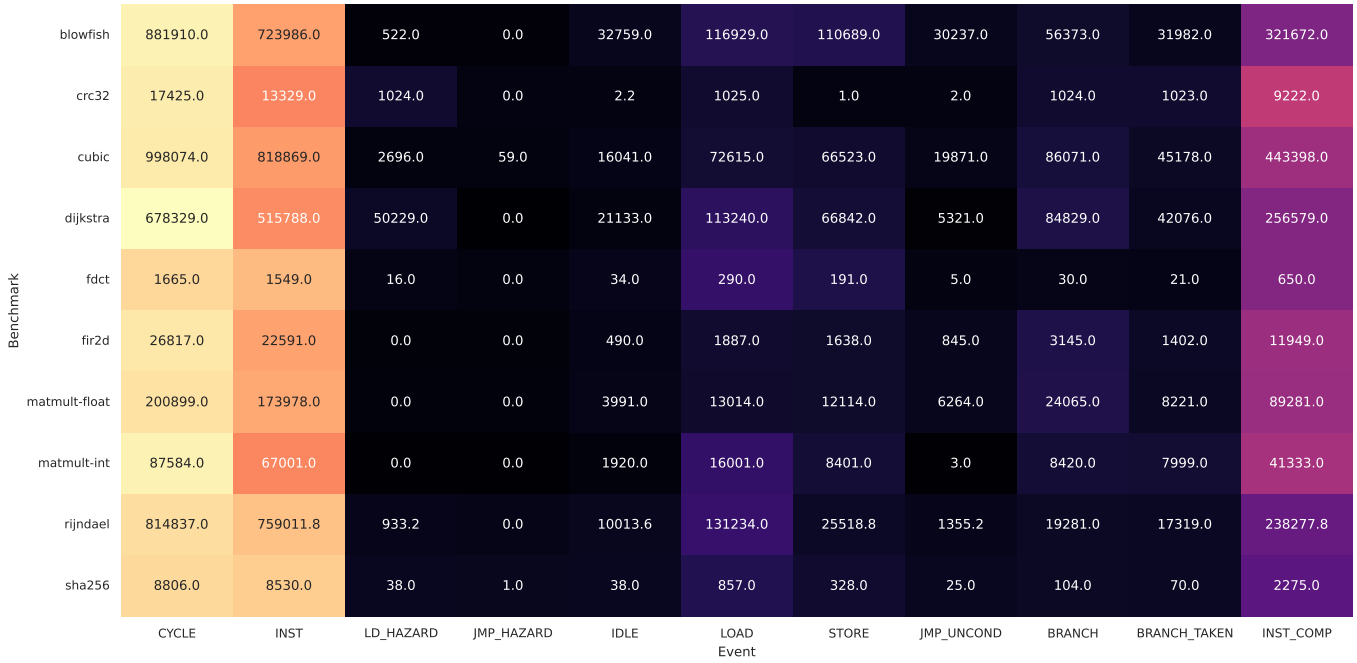


Figure 4.3 – Heatmap of mean event occurrence per benchmark, with colors normalized by benchmark (row)

Figure 4.4 – Relationships between a given `mpcer` event's frequency and the respective sample's mean power consumption.

mpcerBitmask benchmark	1	2	4	8	16	32	64	128	256	512	1024
blowfish	0	0	0	0	0	0	0	0	0	0	0
crc32	0	0	0	0	422.95×10^{-3}	0	0	0	0	0	0
cubic	0	0	0	0	0	0	0	0	0	0	0
dijkstra	0	0	0	0	0	0	0	0	0	0	0
fdct	0	0	0	0	0	0	0	0	0	0	0
fir2d	0	0	0	0	0	0	0	0	0	0	0
matmult-float	0	0	0	0	0	0	0	0	0	0	0
matmult-int	0	0	0	0	0	0	0	0	0	0	0
rijndael	95.88	6.53	16.48	0	35.51	0	1.31	435.19×10^{-3}	0	0	3.05
sha256	0	0	0	0	0	0	0	0	0	0	0

Table 4.2 – Standard deviations of events per benchmark and `mpcer` bitmask. For a mapping of bitmasks to events, see register diagram 3.1.

In contrast Figure 4.4 shows how `JMP_UNCOND` and `BRANCH` operations are associated with a strong *drop* in mean current, as these deviate from the processor's previous control flow, leading to possible pipeline stalls and subsequent bubbling, which may be supported by the negative correlation of Energy with `CYCLE` events. As the ESP32-C3 does not employ a branch predictor, it is no surprise that the latter event has an even more drastic effect on power consumption. In contrast, `BRANCH_TAKEN` has a lesser correlation with power, as the processor begins following a new sequence of instructions and must insert NOPs. The weak `INST_COMP` event correlation is likely due to the instruction cache (I-cache). Although compressed instructions are a feature originally implemented to reduce RISC binary code size, the reduced instruction fetch- and decoding efforts provide a mild performance gain [Yos+97; LHW00], especially in devices with an I-cache, such as the ESP32-C3 [CBM97; Esp24c].

The observed trends in the data suggest *a priori* that a linear model will effectively capture the underlying relationships between the variables.

4.3 Model Selection and Validation

There is a real possibility that a model may overfit, in other words, a known combination of ticks, bitmask, and event count may be identified, especially as many samples are gathered for each `mpcer` bitmask for each benchmark. To illustrate the effectiveness of a model for general data, foreign samples from benchmarks not included in the training set are employed to assess a model's performance. As such, the same procedure described in subsection 3.3.3 is repeated with other benchmarks found in the BEEBS code repository, namely "whetstone" (a commonly-employed benchmark dating to 1976), "levenstein" (a benchmark measuring string differences, known as the Levenshtein distance), "jfdct" (JPEG's discrete cosine transform), "jannes-complex" (particle flow analysis) and "huffbench" (a common compiler benchmark). These benchmarks are also available as part of the BEEBS project and were chosen as candidates for validation due to their ubiquity or historical prominence [PHB13].

The resulting data from the ideal ten BEEBS benchmarks for modelling power is then used to fit several linear models available from "Scikit-learn: Machine Learning in Python," which must then predict the current from the five other BEEBS benchmarks selected for validation.

4.3.1 Determining Model Accuracy with Metrics and Cross-Validation

Various metrics have been developed to score the error between the actual ($\boldsymbol{y}$) and predicted ($\hat{\boldsymbol{y}}$) values of linear regression models. The perhaps most straight-forward metric, the mean absolute error (MAE), measures the mean error (difference) between all n predicted values and actual values, as seen in Equation (4.2). In its simplicity lies its strength: it retains the units of the prediction (e.g. mA) and treats statistical outliers no different from other values. Like the coefficient of variation (CV), it is also easy to represent as a percentual value relative to the largest actual value, then referred to as the mean absolute percentage error (MAPE), Equation (4.3), making it sensitive to *relative* errors in the predicted values [Ped+11]. As both of these metrics measure the error, a low value is most desirable.

$$\mathrm{MAE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{4.2}$$

$$\mathrm{MAPE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n}\sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{\max |y_i|} \tag{4.3}$$

A more complex but expressive metric is the coefficient of determination (R^2), which "provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of explained variance." [Ped+11] Equation (4.4) shows a definition of R^2 as used by *scikit-learn*, with the mean $\bar{y}$ defined as $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ [Ped+11].

$$R^2(\boldsymbol{y}, \hat{\boldsymbol{y}}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4.4}$$

Note that, because the metric subtracts from 1, always predicting the mean value (i.e. a flat line across the trend) would result in a score of $1 - \frac{1}{1} = 0$, and predicting *against* the correlation of data (e.g. a downwards line, when data points trend upwards) causes the value of the fraction to increase, resulting in a *negative* score. Positive scores therefore indicate more accurate predictions, with the best possible score being 1 if *all* the predicted trend line *perfectly* intercepts all data points. The resulting metric can, because of its proportional nature, be represented as a percentage.

A flaw of the R^2 is that it may increase as insignificant features are added. In the 1929 publication "The Application of the Theory of Error to Multiple and Curvilinear Correlation," Ezekiel proposed the *adjusted coefficient of determination ($\overline{R}^2$)* shown in Equation (4.5), which accounts for the number of predictors (p) in a linear regression model trained on n samples, making it a stronger indicator of the model's explanatory power relative to its complexity, especially useful when comparing models [Eze29].

$$\overline{R}^2 = 1 - (1 - R^2) \cdot \frac{n-1}{n-p-1} \tag{4.5}$$

Because the discrepancy between the training targets of the benchmarks is only a few mA, the MAE/MAPE metrics will yield comparably small scores, and because the quality of the regression plays a deciding factor in this case, the $\overline{R}^2$ metric serves as the main criterion to determine if a model fits the trends in events better than simply predicting the mean.

4.3.2 Comparison of Linear Models

The OLS method of determining the optimal weights of the linear model described in Chapter 2 is commonly extended using hyperparameters (tunable coefficients independent of training

data, set before the solving process begins) and regularization (controlling the relative strength of feature weights to prevent overfitting) to further modify the weighting of features. The alternatives and their advantages are briefly presented and compared.

- The *Ridge* regression by Hoerl and Kennard extends OLS "by imposing a penalty on the size of the coefficients" [Ped+11], known as the ℓ_2 penalty. It uses the hyperparameter $\alpha \geq 0$ to control the learning rate, adding robustness against highly collinear[17] features [HK70; Ped+11], which can make it difficult to determine the individual effects of a feature.
- The *least absolute shrinkage and selection operator (Lasso)* regression by Tibshirani again extends OLS with an α, however drives the coefficient vector towards an ℓ_1 norm instead, thereby strongly reducing or even completely eliminating the weights of less-relevant features [Tib96; Ped+11].
- The *elastic net* regression by Zou and Hastie combines the ℓ_1 and ℓ_2 normalizations. This allows for the strong feature selection of *Lasso*, picking multiple features if these correlate with one another [ZH05; Ped+11].

A variety of preprocessing and training methods were employed in an effort to create a model with a high degree of predictive accuracy.

4.3.2.1 Training Using a Preprocessing Pipeline

The initial attempts to train a linear model relied on directly fitting a model using *scikit-learn*'s Pipeline module to pre-process feature data. Before the model is *fit* (trained) on data or used to make predictions, the operations seen in Figure 4.5 are applied on each column of the collected samples.

The `mpcer` bitmask is treated separately from other data points of a sample, as it represents nominal, not ordinal, data. It is used to *one-hot* encode the data, essentially separating each sample's data by the `mpcer` bitmask via a binary encoding scheme [Ped+11]. By calculating the second polynomial, interactions between inputs can better be observed, in this case beneficial as it has already been established in Section 4.1 that a relationship exists between the number of times an event is observed and the duration of a benchmark. This extended set of features is then normalized column-wise as to ensure that all inputs are on a similar scale when multiplied with their respective weights.

This transformed input matrix was then used to fit a model based on *scikit-learn's* `ElasticNetCV` model. As implied by the name, it is based upon the elastic net regression by Zou and Hastie, with this specific implementation automatically selecting the ideal coefficients for its ℓ_1 and ℓ_2 normalization by means of cross-validation (CV), in which a certain ratio of training data is reserved and compared to predicted values, adjusting hyperparameters thereafter if the R^2 score worsens [ZH05; Ped+11].

The final model resulted in an error of only 1.85 % MAPE (604.56 μA MAE), but an R^2 score of -0.01, a strong indicator that it is ineffective by the criteria previously set in Section 4.3.1.

4.3.2.2 Training on Aggregated Data

Due to the theoretically promising yet unsuccessful attempts at training a model using the pipeline-transform method, a different approach is needed, using the pivot-normalization

[17] The issue of *multicollinearity* arises when when two or more independent variables (i.e. PMC event frequencies) are similarly correlated with the dependent variable (i.e. power consumption)

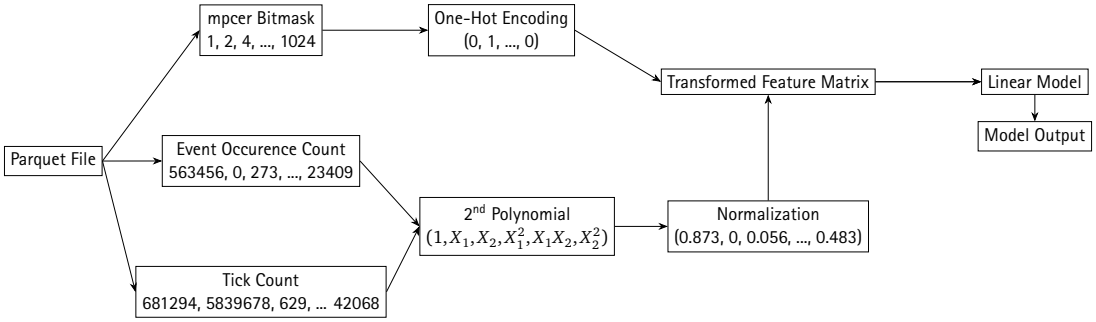


Figure 4.5 – Feature preprocessing pipeline for a linear model.

method described in Section 2.4.2 of Chapter 2 and already implemented for Figure 4.4. An exhaustive search of all $1 < p < 11$ combinations of events is used to train and score a model to determine the highest scoring combinations in a process called *feature selection,* from which a pivot table with a moving window over each of the $n = 100$ samples taken per benchmark/`mpcer` bitmask is created and normalized. Note that the effects of normalization are much more pronounced for each event with a smaller p, as only a subset of the total counts are now included.

These proportional events are then used as direct training features for *scikit-learn*'s *LinearRegression, RidgeCV, LassoCV,* and *ElasticNetCV* models. The three top scoring models are shown in Table 4.3 — interestingly, each was an instance of *LinearRegression*.

Composite `mpcer` events in aggregation	$\overline{R}^2$	R^2	MAPE	MAE
`INST|BRANCH_TAKEN|INST_COMP`	0.619 020	0.619 402	0.88 %	283 μA
`INST|LD_HAZARD|IDLE|BRANCH_TAKEN|INST_COMP`	0.555 249	0.555 694	1.08 %	346 μA
`CYCLE|BRANCH_TAKEN|INST_COMP`	0.491 844	0.492 353	1.25 %	405 μA

Table 4.3 – Three top-scoring models after an exhaustive feature- and hyperparameter search. Note that all are instances of *LinearRegression*.

Given the large amount of noise in Figure 4.4 and discrepancies in the CPEs, an $\overline{R}^2 > 0.5$ can be interpreted as moderate to substantial [HRS09; Nau20], as the majority of variance can be explained. It can thus be confidently concluded that the highest scoring linear model uses the counted *regular* as well as *compressed instructions* and *branches taken* to effectively predict current, and thus power, consumption.

4.4 Extending Events to an RTOS

The next reasonable step in evaluating the power characteristics of a computer system is to measure a piece of software with more complex behavior than the mostly deterministic benchmarks; in the case of an embedded system specifically, an RTOS would be a suitable environment.

Because the ESP32-C3 does *not* feature the atomic (`A`) ISA extension, several steps must be taken to ensure that shared resources can safely be synchronized; the ISA specification notes that for `RV32I`, "RISC-V does not guarantee that stores to instruction memory will be made

visible to instruction fetches on the same RISC-V thread until a FENCE.I instruction is executed" [Wat+19], in which "... a simple implementation ... might be able to implement the FENCE and FENCE.I instructions as NOPs." [Wat+19] The *ESP32-C3 Technical Reference Manual* also recommends that, before the global machine mode interrupt enable (`MIE`) is enabled, a `FENCE` instruction[18] must be executed [Esp24c]. This overhead will be *a priori* measurable in the `CYCLE`, `INST` and potentially `IDLE` counts when an RTOS manages interrupts, CPU time, and hardware resources.

The Zephyr RTOS includes a tracing framework which implements the common trace format (CTF) and, when enabled, can can automatically log kernel as well as application events [Zi23]. The major advantage over simply counting PMC data is that the RTOS can multiplex data; even on a simple processor, is is feasible to count multiple software events. Similar to the transmission process described in Chapter 3, CTF serves the purpose of serializing and transmitting event data to a host, but supports concrete data types, including maps (key-value pairs) and full data structures [Pro24, Section 5.3].

Zephyr integrates its tracing functionality to several host-side tools, including Percepio AB's *Tracealyzer* and the *Eclipse Trace Compass*, but also allows users to directly define custom event handlers[19] for thread scheduling, interrupt service routines (ISRs), and idling [Zi23]. Deeper tracing at the kernel level is also possible via the `zephyr/tracing/tracing.h` header file, which features a plethora of additional tracing APIs. Those expected to correlate with energy based on high instruction counts, branching, and memory accesses are:

Semaphores and Mutexes are used to count and limit access to specific resources in both the kernel- and userspace of Zephyr. Not only will they cause higher instruction counts on an processor such as the ESP32-C3, but their attempted and blocked entries can also be traced, metrics which likely correlate with system and interrupt activity;

Memory Slabs are kernel objects used by the RTOS to dynamically allocate memory blocks from specific regions. Especially of interest is the blocking behavior[20], as each memory slab must access must be synchronized;

Syscalls in the RTOS are, by very definition, likely to cause significant branching, as the single-threaded CPU must begin executing a new code path; syscall tracing has often been used in power modelling applications for embedded systems [Pat+11; Agg+14].

4.5 Discussion

One of the difficulties encountered in training were the extremely small measurement ranges. Although trend lines in Figure 4.4 show noticeable trends in data for common events such as `LOAD`, `STORE`, `JMP_UNCOND`, `BRANCH` and `INST_COMP`, unfortunately, the dataset is still quite compressed; frequencies range only over 5 %–10 % with circa 5 mW power difference with significant noise relative to the size. This noise is likely a significant contributor to the R^2 score described in Section 4.3.2.2. An architecture geared towards more sensitivity *may* have yielded better results; as the finest scale for the HMO3000-Series oscilloscope is limited to 1 mV per div, a smaller shunt resistor value would yield larger voltage-drop readings. For comparison, Walker et al. were able

[18] "The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V threads and external devices or coprocessors" [Wat+19]

[19] `https://docs.zephyrproject.org/latest/services/tracing/index.html#user-defined-tracing`

[20] `sys_port_trace_k_mem_slab_alloc_blocking(slab, timeout)`

to achieve an R^2 of 0.99 on ARM Cortex-A7 and A15 CPUs [Wal+17]. In response, other kinds of non-linear tree-based regression models were attempted to be fit, ranging from plain decision trees to ensembled histogram-based gradient boosting regression trees, again with exhaustive feature set searches. The only one which performed better than the simple OLS linear regression was the *HistGradientBoostingRegressor* using `CYCLE|INST|IDLE|STORE|BRANCH|BRANCH_TAKEN|INST_COMP`, which scored an $\overline{R}^2$ of 0.73, albeit with a slightly higher MAPE of 0.93 %. This regressor, while effective, is outside of the scope of this thesis and may serve as an interesting starting point for future work.

While the MAPEs of the models trained in this thesis are satisfactory ($\leq 2\,\%$) and correspond to the findings in related power-modelling work, it is difficult to compare the *quality* of regression without the R^2, which was rarely published [Rod+13; Pat+11; Nik+21; Geo+21]. The strong variations in CPE shown in Figure 4.2 suggest that events such as `JMP_UNCOND` and `LD_HAZARD` are strongly dependent on the processor state, and indicate that a larger pool of benchmarks may increase the precision of future models, which may also profit from stochastic approaches that consider the ordering of events.

One further unexplored topic is the effect of compiler flags on the energy characteristics of the CPU. In the interest balancing a production use case where performance efficiency takes precedence over debuggability, while avoiding I-cache misses and undefined behavior [Myt+09; DB17], all BEEBS benchmarks passed to *espbench* have been compiled with *c3dk*'s default `-O2` GCC flag, in contrast to other PMC-oriented power modelling experiments of Singh, Bhadauria, and McKee; Mair et al.

5 CONCLUSION

Power modelling has played and will continue to play an important role in the design and analysis of both embedded software and hardware as more attention is paid to optimizing the resource usage and lifetime of embedded systems.

By preparing a processor for interference-free bare-metal benchmarking and running a set of diverse benchmarks, it was possible to identify small but useful correlations between the frequency of occurrence in an event and energy consumption. In the past, these benchmarks were explicitly selected for the measurement of power on embedded systems, and still provided a fair range of performance monitoring counter (PMC) data to train a diverse set of linear models. Metrics were introduced to assess the goodness of the model estimates, where, in contrast to some previous power modelling publications, a focus was placed on not just minimizing the mean absolute error (MAE), but also maximizing the adjusted coefficient of determination ($\bar{R}^2$) score, ensuring that the regression model explained as much variance as possible. The integration of RTOS traces will likely further increase the precision of modelling to an even greater extent. Insight into these trace metrics not only opens up more possibilities for power modelling, but also optimizations in user application code as well as well as kernel configuration.

The models used nothing but the aggregated PMC data already available to the processor itself. As such, it is not ruled out that the models, given their relative simplicity, could be pre-trained and either used on a coprocessor such as an FPGA, in an approach similar to that of Nikov et al. [Nik+21], or run directly on RISC-V based SoCs using projects such as TinyML[21] and its related projects, which are specifically tailored to machine learning applications on embedded systems [Lin+23].

[21] https://hanlab.mit.edu/projects/tinyml

LIST OF ACRONYMS

$\overline{R}^2$ adjusted coefficient of determination

σ standard deviation

R^2 coefficient of determination

`MIE` global machine mode interrupt enable

`mpccr` machine performance counter count register

`mpcer` machine performance counter event register

`mpcmr` machine performance counter mode register

ABI application binary interface

API application programmer interface

CISC complex instruction set computer

CPE cycles per event

CPU central processing unit

crc32 cyclic redundancy check, 32-bit

CSR control and status register

CSV comma-seperated values

CTF common trace format

CV coefficient of variation

CV cross-validation

DC direct current

DRAM data RAM

DWT Data Watchpoint Trace

ELF Executable and Linkable Format

ESP-IDF Espressif IoT Development Framework

FDCT forward discrete cosine transform

FPGA field programmable gate array

GCC GNU compiler collection

GNU "GNU's not UNIX" project

GPIO general-purpose input/output

HAL hardware abstraction layer

HPC high-performance computing

I-cache instruction cache

I/O input/output

IC integrated circuit

IoT internet of things

IRAM instruction RAM

ISA instruction set architecture

ISR interrupt service routine

JTAG Joint Test Action Group (debugging standard named after group)

Lasso least absolute shrinkage and selection operator

LDO low-dropout voltage regulator

MAE mean absolute error

MAPE mean absolute percentage error

matmult-int integer matrix multiplication

ML machine learning

NOP no-operation

OLS ordinary least squares

OS operating system

PLL phase-locked loop

PMC performance monitoring counter

PSU power supply unit

RAM random access memory

RAPL running average power limit

RISC reduced instruction set computer

RNG random number generation

ROM read-only memory

RTOS real-time operating system

SDK software development kit

SHA256 Secure Hash Algorithm 2, 256 bit digest

SoC system on a chip

SPI serial peripheral interface

UART universal asynchronous receiver transmitter

USB universal serial bus

USBTMC USB test & measurement class

Xtal crystal oscillator

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[Agg+14] Karan Aggarwal et al. “The power of system call traces: predicting the software energy consumption impact of changes.” In: *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*. CASCON ’14. Markham, Ontario, Canada: IBM Corp., 2014, pp. 219–233.

[And15] Gabor Andai. “Performance monitoring on high-end general processing boards using hardware performance counters.” MA thesis. KTH, School of Information and Communication Technology (ICT), 2015, p. 68.

[ARM24] ARM Limited. *ARM® Cortex®-M33 Processor. Technical Reference Manual*. Version r1p0. 2024. 137 pp. URL: https://developer.arm.com/Processors/Cortex-M33 (visited on 09/28/2024).

[Arn24] Johannes Karl Arnold. *c3dk. A bare metal SDK for the ESP32-C3*. Version ff0e77a. 2024. URL: https://github.com/j0hax/c3dk (visited on 08/16/2024).

[CBM97] I-Cheng K. Chen, Peter L. Bird, and Trevor N. Mudge. *The Impact of Instruction Compression on I-cache Performance*. Tech. rep. CSE-TR-330-97. EECS Department, University of Michigan, 1997. URL: https://api.semanticscholar.org/CorpusID:860126.

[Dav+10] Howard David et al. “RAPL: memory power estimation and capping.” In: *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED ’10. Austin, Texas, USA: Association for Computing Machinery, 2010, pp. 189–194. ISBN: 9781450301466. DOI: 10.1145/1840845.1840883. URL: https://doi.org/10.1145/1840845.1840883.

[DB17] Manjeet Dahiya and Sorav Bansal. “Black-Box Equivalence Checking Across Compiler Optimizations.” In: *Programming Languages and Systems*. Ed. by Bor-Yuh Evan Chang. Cham: Springer International Publishing, 2017, pp. 127–147. ISBN: 978-3-319-71237-6.

[Den+23] Eva Dengler et al. “FusionClock: Energy-Optimal Clock-Tree Reconfigurations for Energy-Constrained Real-Time Systems.” In: *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Ed. by Alessandro V. Papadopoulos. Vol. 262. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 6:1–6:23. ISBN: 978-3-95977-280-8. DOI: 10.4230/LIPIcs.ECRTS.2023.6. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2023.6.

[Don] Dongguan Korad Technology Co., Ltd. *Programmable DC Power Supply. KD3000-6000 Series User Manual*. 6 pp. URL: https://v4.cecdn.yun300.cn/site_1801250020/KD3000--6000%20Series%20User%20Manual%20V1.12.pdf (visited on 10/06/2024).

[Esp24a] Espressif Systems (Shanghai) Co., Ltd. *ESP32-C3-DevKitM-1*. Version 5.3. 2024. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html (visited on 06/30/2024).

[Esp24b] Espressif Systems (Shanghai) Co., Ltd. *ESP32-C3-MINI-1 Datasheet*. Version 1.5. 2024. Chap. 4, p. 15. 35 pp. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3-mini-1_datasheet_en.pdf (visited on 07/11/2024).

[Esp24c] Espressif Systems (Shanghai) Co., Ltd. *ESP32-C3 Technical Reference Manual*. Version 1.1. 2024. Chap. 1, pp. 28–56. 877 pp. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf (visited on 06/30/2024).

[Eze29] Mordecai Ezekiel. "The Application of the Theory of Error to Multiple and Curvilinear Correlation." In: *Journal of the American Statistical Association* 24.165 (1929), pp. 99–104. ISSN: 01621459, 1537274X. URL: http://www.jstor.org/stable/2277015 (visited on 10/12/2024).

[Fur+22] Gianluca Furano et al. "A European Roadmap to Leverage RISC-V in Space Applications." In: *2022 IEEE Aerospace Conference (AERO)*. 2022, pp. 1–7. DOI: 10.1109/AERO53065.2022.9843361.

[Geo+21] Kyriakos Georgiou et al. "A Comprehensive and Accurate Energy Model for Arm's Cortex-M0 Processor." In: (2021). arXiv: 2104.01055 [cs.SE]. URL: https://arxiv.org/abs/2104.01055.

[HK70] Arthur E. Hoerl and Robert W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." In: *Technometrics* 12.1 (1970), pp. 55–67. ISSN: 00401706. URL: http://www.jstor.org/stable/1267351 (visited on 10/09/2024).

[HRS09] Jörg Henseler, Christian Ringle, and Rudolf Sinkovics. "The Use of Partial Least Squares Path Modeling in International Marketing." In: vol. 20. Jan. 2009, pp. 277–319. ISBN: 9781848554689. DOI: 10.1108/S1474-7979(2009)0000020014.

[Lee+01] Sheayun Lee et al. "An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors." In: *SIGPLAN Not.* 36.8 (2001), pp. 1–10. ISSN: 0362-1340. DOI: 10.1145/384196.384201. URL: https://doi.org/10.1145/384196.384201.

[Lee+23] Joseph K. L. Lee et al. "Test-Driving RISC-V Vector Hardware for HPC." In: *High Performance Computing*. Ed. by Amanda Bienz et al. Cham: Springer Nature Switzerland, 2023, pp. 419–432. ISBN: 978-3-031-40843-4.

[Leg06] Adrien Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes. Avec un supplément contenant divers perfectionnemens de ces méthodes et leur application aux deux comètes de 1805*. French. Courcier, 1806.

[LHW00] H. Lekatsas, J. Henkel, and W. Wolf. "Code compression for low power embedded system design." In: *Proceedings 37th Design Automation Conference*. 2000, pp. 294–299. DOI: 10.1145/337292.337423.

[Lin+23] Ji Lin et al. "Tiny Machine Learning: Progress and Futures [Feature]." In: *IEEE Circuits and Systems Magazine* 23.3 (2023), pp. 8–34. DOI: 10.1109/MCAS.2023.3302182.

[Lyu22] Sergey Lyubka. *MDK. A bare metal SDK for the ESP32 & ESP32C3*. Version 39a23c4. Cesanta Software Ltd., 2022. URL: https://github.com/cpq/mdk (visited on 08/10/2024).

[Mai+13] Jason Mair et al. "Myths in PMC-Based Power Estimation." In: *Energy Efficiency in Large Scale Distributed Systems*. Ed. by Jean-Marc Pierson, Georges Da Costa, and Lars Dittmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 35–50. ISBN: 978-3-642-40517-4.

[MR+23] Roberto Molina-Robles et al. "An Energy Consumption Benchmark for a Low-Power RISC-V Core Aimed at Implantable Medical Devices." In: *IEEE Embedded Systems Letters* 15.2 (2023), pp. 57–60. ISSN: 1943-0671. DOI: 10.1109/LES.2022.3190063.

[Myt+09] Todd Mytkowicz et al. "Producing wrong data without doing anything obviously wrong!" In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: Association for Computing Machinery, 2009, pp. 265–276. ISBN: 9781605584065. DOI: 10.1145/1508244.1508275. URL: https://doi.org/10.1145/1508244.1508275.

[Nau20] Robert Nau. *What's a good value for R-squared?* Statistical forecasting: notes on regression and time series analysis. Fuqua School of Business. 2020. URL: https://people.duke.edu/~rnau/rsquared.htm (visited on 10/12/2024).

[Nik+21] Kris Nikov et al. "Robust and Accurate Fine-Grain Power Models for Embedded Systems With No On-Chip PMU." In: *IEEE Embedded Systems Letters* 14 (2021), pp. 147–150. URL: https://api.semanticscholar.org/CorpusID:235266247.

[NKK04] H.T. Nguyen, L.M. King, and G. Knight. "Real-time head movement system and embedded Linux implementation for the control of power wheelchairs." In: *The 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. Vol. 2. 2004, pp. 4892–4895. DOI: 10.1109/IEMBS.2004.1404353.

[Pat+11] Abhinav Pathak et al. "Fine-grained power modeling for smartphones using system call tracing." In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: Association for Computing Machinery, 2011, pp. 153–168. ISBN: 9781450306348. DOI: 10.1145/1966445.1966460. URL: https://doi.org/10.1145/1966445.1966460.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PHB13] James Pallister, Simon Hollis, and Jeremy Bennett. "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms." In: *arXiv e-prints*, arXiv:1308.5174 (Aug. 2013), arXiv:1308.5174. DOI: 10.48550/arXiv.1308.5174. arXiv: 1308.5174 [cs.PF].

[Pro24] Philippe Proulx. *CTF2 SPEC 2.0. Common Trace Format version 2*. Tech. rep. Version 2.0. DiaMon Workgroup, 2024. URL: https://diamon.org/ctf/ (visited on 10/12/2024).

[RIG24] RIGOL TECHNOLOGIES CO., LTD. *RIGOL Data Sheet. DM3058/DM3058E Digital Multimeter*. 2024. 8 pp. URL: https://beyondmeasure.rigoltech.com/acton/attachment/1579/f-001f/0/-/-/-/-/file.pdf (visited on 07/11/2024).

[Rod+13] Rance Rodrigues et al. “A Study on the Use of Performance Counters to Estimate Power in Microprocessors.” In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 60.12 (2013), pp. 882–886. DOI: 10.1109/TCSII.2013.2285966.

[SBM09] Karan Singh, Major Bhadauria, and Sally A. McKee. “Real time power estimation and thread scheduling via performance counters.” In: *SIGARCH Comput. Archit. News* 37.2 (July 2009), pp. 46–55. ISSN: 0163-5964. DOI: 10.1145/1577129.1577137. URL: https://doi.org/10.1145/1577129.1577137.

[SP24] Leonhard Stiny and Martin Poppe. “Der unverzweigte Gleichstromkreis.” German. In: *Grundwissen Elektrotechnik und Elektronik: Eine leicht verständliche Einführung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024, pp. 23–55. ISBN: 978-3-662-68459-7. DOI: 10.1007/978-3-662-68459-7_2. URL: https://doi.org/10.1007/978-3-662-68459-7_2.

[Spa17] Philip Sparks. *The route to a trillion devices. The outlook for IoT investment to 2035*. White Paper. ARM Limited, 2017. URL: https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-1996-00-00-00-01-30-09/ARM-_2D00_-The-route-to-a-trillion-devices-_2D00_-June-2017.pdf.

[Sys24] Espressif Systems. *esptool*. Version v4.8.1. Sept. 2024. URL: https://github.com/espressif/esptool.

[tea24] The pandas development team. *pandas-dev/pandas: Pandas*. Version v2.2.3. Sept. 2024. DOI: 10.5281/zenodo.13819579. URL: https://doi.org/10.5281/zenodo.13819579.

[Tib96] Robert Tibshirani. “Regression Shrinkage and Selection via the Lasso.” In: *Journal of the royal statistical society series b-methodological* 58 (1996), pp. 267–288. URL: https://api.semanticscholar.org/CorpusID:16162039.

[VO22] Timon Van Overveldt. *Counting CPU cycles on ESP32-C3 and ESP32-C6 microcontrollers*. 2022. URL: https://ctrlsrc.io/posts/2023/counting-cpu-cycles-on-esp32c3-esp32c6/ (visited on 10/06/2024).

[Voh16] Deepak Vohra. “Apache Parquet.” In: *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Berkeley, CA: Apress, 2016, pp. 325–335. ISBN: 978-1-4842-2199-0. DOI: 10.1007/978-1-4842-2199-0_8. URL: https://doi.org/10.1007/978-1-4842-2199-0_8.

[Wal+17] Matthew J. Walker et al. “Accurate and Stable Run-Time Power Modeling for Mobile and Embedded CPUs.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.1 (2017), pp. 106–119. DOI: 10.1109/TCAD.2016.2562920.

[Wat+19] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*. Tech. rep. Version 20191214-draft. EECS Department, University of California, Berkeley, 2019. URL: https://riscv.org/technical/specifications/.

[XLT24] Yixiao Xing, Yixiao Li, and Hiroaki Takada. “A Multi-core RTOS Benchmark Methodology To Assess System Services Under Contentions.” In: *Journal of Information Processing* 32 (2024), pp. 829–843. DOI: 10.2197/ipsjjip.32.829.

[Yos+97] Y. Yoshida et al. “An object code compression approach to embedded processors.” In: *Proceedings of 1997 International Symposium on Low Power Electronics and Design*. 1997, pp. 265–268. DOI: 10.1145/263272.263349.

[ZH05] Hui Zou and Trevor Hastie. “Regularization and Variable Selection via the Elastic Net.” In: *Journal of the Royal Statistical Society. Series B (Statistical Methodology)* 67.2 (2005), pp. 301–320. ISSN: 13697412, 14679868. URL: http://www.jstor.org/stable/3647580 (visited on 10/09/2024).

[Zi23] Zephyr Project members and individual contributors. *Zephyr Project Documentation*. Tech. rep. Version 3.7.99. Zephyr Project, 2023. URL: https://docs.zephyrproject.org/latest/ (visited on 06/12/2024).