POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A feasibility study for an energy prediction and optimization framework

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Alessandro Bertulli**

Student ID: 944699
Advisor: Prof. Giovanni Agosta
Co-advisors: Daniele Cattaneo
Academic Year: 2022-23

Abstract

In today's world it has become of undelayable importance to reduce the amount of energy wasted in all human activities. A sector that may benefit from energy saving is information technology, with respect to the amount of energy required to perform a computation (in the broader sense of the word). In this work we investigate the feasibility of building a compiler driver that optimizes its produced code, such that when being executed on a physical machine, it will consume the least possible amount of energy. We explore the basic components needed to build such a driver, and we highlight that it is of great importance to have an accurate oracle capable of predicting the power consumption of an arbitrary program. Moreover, we note that such an oracle must be trained to the specific processor we wish to emit code for: in this work, we focus ourselves on building this specific component. We explain the design choices we made, the steps we performed to measure and profile a test processor (an ARM Cortex-M4), how we generated the test cases, and how we processed the results to build the oracle, using multiple linear regression models. Then, we assess the quality of the resulting predictor, both with usual statistical goodness indicators and by validating it with real programs, also by testing it with the experimental compiler optimization framework Taffo. We show that our predictor reached an accuracy peak of 86.21 %. In the end, we summarise all the difficulties we encountered in our example predictor building process, and we outline the requirements needed to build accurate oracles.

Keywords: power, energy estimation, optimization, compilers

Abstract

Nel mondo di oggi è diventato di irrimandabile importanza ridurre la quantità di energia che viene sprecata in tutte le attività umane. Un settore che può trarre beneficio dal risparmio energetico è l'informatica, riferendoci alla quantità di energia richiesta per effettuare un calcolo (nel senso più ampio del termine). In questa tesi indaghiamo la fattibilità di sviluppare un compilatore che ottimizzi il proprio codice prodotto, in modo che quando eseguito sulla macchina target, consumi la minor quantità di energia possibile. Esploriamo i componenti base necessari per costruire un simile compilatore, e sottolineiamo che è di grande importanza avere un oracolo accurato capace di predire il consumo in potenza di un programma arbitrario. Inoltre, notiamo che un simile oracolo deve essere addestrato sul specifico processore per cui vorremmo emettere codice: in questa tesi, ci concentriamo sulla costruzione di questo specifico componente. Spieghiamo le scelte di design che abbiamo fatto, i passi che abbiamo eseguito per misurare e profilare un processore di prova (un ARM Cortex-M4), come abbiamo generato i casi di test, e come abbiamo processato i risultati per costruire l'oracolo, usando modelli lineari a regressione multipla. Dopodiché, valutiamo la qualità del predittore risultante, sia con i soliti indicatori statistici di bontà sia validandolo con programmi reali, anche testandolo con il framework di ottimizzazione per compilatori sperimentale Taffo. Mostriamo che il nostro oracolo raggiunge picchi di accuratezza del 86.21 %. Infine, ricapitoliamo tutte le difficoltà che abbiamo incontrato nella costruzione del nostro predittore d'esempio, e delineiamo i requisiti necessari per costruire oracoli accurati.

Parole chiave: potenza, stima energetica, ottimizzazione, compilatori

Contents

Contents iii

List of Tables v

List of Figures vii

List of Listings ix

1 Introduction 1
- 1.1 The importance of energy estimation . . . 1
- 1.2 The ideal tool . . . 2
- 1.3 Thesis structure . . . 3

2 State of the art 5
- 2.1 Estimation techniques . . . 5
- 2.2 Previous works on ISA models . . . 6
  - 2.2.1 General hypotheses . . . 6
  - 2.2.2 Ideal power estimation . . . 8
  - 2.2.3 Using functions of the code . . . 10
  - 2.2.4 The Dual Bit Type method . . . 11
  - 2.2.5 Precise multi-threading processor estimation . . . 14
- 2.3 Conclusions . . . 15

3 Power model construction 19
- 3.1 Experimental components . . . 19
  - 3.1.1 The processor . . . 19
  - 3.1.2 The measurement tool . . . 19
  - 3.1.3 The tentative mathematical model . . . 20
  - 3.1.4 The measurement workflow . . . 22
  - 3.1.5 The train dataset generation . . . 25

3.1.6 Memory instructions 27
3.1.7 Electrical wiring 28
3.1.8 Data analysis 31
3.1.9 Instruction interleaving 35
3.1.10 Trace selection 39

4 Experimental results **43**
4.1 Data exploration 43
4.1.1 Assessing the regression variables effects 43
4.1.2 The binary weight dependency 52
4.2 Results 54
4.2.1 Goodness of the linear models 54
4.2.2 Model validation 57

5 Conclusion **81**
5.1 Model results 81
5.2 Model feasibility 82

A Instruction generation code **85**

B Data analysis code **89**

C Predictor code **93**
C.1 Single instruction 93
C.2 Multiple instructions 95

Bibliography **97**

List of Tables

2.1 Summary comparison of the power estimation techniques. 7
2.2 Summary comparison of the analysed previous works. 17

3.1 Otii Arc Pro precisions 20

4.1 Filters applied to isolate the variable effects. 45
4.2 Summary of statistics changes for the forwarding paths. 46
4.3 Summary of statistics changes for the APSR usage. 47
4.4 Summary of statistics changes for the conditional execution. 49
4.5 Summary of statistics changes for using an immediate operand. 50
4.6 Results fact recursive with QEMU 62
4.7 Results for some Taffo benchmarks. 78
4.8 Results for the Taffo optimization process. 79
4.9 Results fact recursive with binary estimation 80

List of Figures

3.1.1 Measurement workflow. 23
3.1.2 Binary weight densities for various steps 26
3.1.3 Effect of the Faraday shield from the 50 Hz disturbance. 29
3.1.4 Circuit scheme of the measurement setup. 30
3.1.5 Call graph for the example program. 42

4.1.1 Effect of the CPU forwarding paths. 44
4.1.2 Effect of updating the APSR 46
4.1.3 Effect of using the conditional execution. 48
4.1.4 Effect of using the barrel shift. 49
4.1.5 Effect of using an immediate value as second operand. 51
4.1.6 Dependency of power by binary weight, register operands 53
4.1.7 Dependency of power by binary weight, immediate operands 55
4.1.8 Dependency of the power consumption on the branch distance. 56
4.2.1 Histogram of error distributions of the linear models. 58
4.2.2 Difference between measured and predicted power values for the linear models. 58
4.2.3 Difference between measured and predicted power values for a few instructions. 59
4.2.4 Heat map measured power, deterministic programs 63
4.2.5 Heat map predicted power, deterministic programs with QEMU 64
4.2.6 Heat map error power, deterministic programs with QEMU 65
4.2.7 Heat map error power (absolute), deterministic programs with QEMU . . 66
4.2.8 Heat map sign error, deterministic programs with QEMU 68
4.2.9 Heat map measured power, Taffo benchmarks 69
4.2.10 Heat map predicted power, Taffo benchmarks 70
4.2.11 Heat map error power, Taffo benchmarks 71
4.2.12 Heat map error power (absolute), Taffo benchmarks 72
4.2.13 Heat map sign error, Taffo benchmarks 73
4.2.14 Heat map predicted power, deterministic programs with binary estimation 74

4.2.15 Heat map error power, deterministic programs with binary estimation . . 75
4.2.16 Heat map error power (absolute), deterministic programs with binary estimation 76
4.2.17 Heat map sign error, deterministic programs with binary estimation . . . 77

List of Listings

3.1 Linear regression model formula. 33
3.2 Reduced linear regression model formula. 34
3.3 Formulas for the inter instruction overhead linear model. 38
3.4 GDB command file when connecting to a QEMU simulated program. . . 40
3.5 Example extract of a GDB log. 41
3.6 Fractions of `objdump` output for the example program. 42

4.1 Deterministic programs. 61

A.1 Cartesian product range code. 86

B.1 Julia code to construct the DataFrame. 90

C.1 Julia code to compute the power consumption of a single instruction. . . 94
C.2 Julia function to predict the power estimate of a sequence of instructions. 96

1 | Introduction

1.1. The importance of energy estimation

Along with a constant search for speed, reliability, composability, today's software engineering aims at a careful use of resources. Non-functional objectives like executing work in an efficient way can be of great importance in some settings. This is true for a variety of resources, like storage space, memory or time, but in this work we focus our analysis on the energetic aspect.

By estimating the amount of energy required to perform a certain calculation (in the broader sense of the term), developers can better estimate the resources required by their systems. For example, embedded developers may predict how long the batteries of their devices will last. This in turn leads to a better planning of the required maintenance interventions. Moreover, data centers around the world usually use large amounts of electrical energy: server administrators can know how much their system will absorb, and they can scale them accordingly.

On the other hand, reducing the amount of energy required to perform a task is important to reduce the environmental footprint digital systems have, a topic of truly undelayable importance nowadays. Moreover, it can reduce the costs of running such systems. For instance, embedded devices may extend the useful lifetime of their batteries, and a reduction in server maintenance costs can be beneficial for the agency running them.

Software has been optimized from almost the beginning of its existence. Usually, this advancement has been driven by the need to reduce the time required by a program, both by researching more efficient algorithms and by using compiler optimizations. Even focusing just on the latter, compiler technology has advanced enormously, providing tools to intelligently inspect and transform code to significantly reduce its execution time. It is reasonable to assume that a software running for less time will require less energy, since the machine running it will be running for less time. This assumption may however be improved in three aspects.

- There are cases in which the fastest code is not necessarily the most efficient. For instance, a sequence of instructions may be replaced with a shorter sequence of more energy consuming ones. To prioritize energy consumption, it would be advisable not to perform the substitution. In other words, energy consumption and speed of execution may be conflicting objective functions.
- There may be sequences of instructions equivalent from the point of view of time, but not of energy. A compiler aiming only to increase time efficiency would simply ignore a transformation not beneficial for its instructed purpose, even if it may bring benefits from other points of view.
- This claim has not yet been thoroughly tested: a lot of literature focus on very specific processors, or test a limited range of applications.

We must note that there may be important trade-offs to be considered. For instance, a more efficient system may be not as fast as the old one, and this may be inconvenient or even critical in situations where speed is a priority, like emergency systems.

1.2. The ideal tool

We hypothesize a compiler tool that can operate energy aware optimizations. Such tool would receive as inputs the source code we would like to compile, along with a flag specifying the target architecture; as a result it would drive the underlying generic compiler to perform a certain number of optimization, so to emit a target code as efficient as possible for that specific architecture. This presents a number of obstacles.

Compiler construction is a delicate science, most of the time non-recyclable between different compilers. As good as the final work would be, it would be then confined to a single compiler. Our aim is instead to apply this research to the maximum number of compilers possible, so to maximize its effectiveness and optimize the work invested in its development. For this reason, we suggest developing for the LLVM compiler technology [14], as it is the backbone of a variety of compilers.

Moreover, in order to know when to perform an optimization, we need to know what is the difference in the predicted power consumption such optimization would introduce. This means computing the difference between the optimization pass input and the output versions of the program. In turn, to do this it is required to be able to estimate the power consumption of an arbitrary program, that is, to build an oracle.

Since we expect different processors to have different power consumptions for the same

program, our oracle needs to be trained with respect to the processor the compiler will emit code for. However, profiling the target architecture is difficult when it is done downstream with respect to the processor manufacturing process. In order to obtain a reliable profile for a given central processing unit (CPU), we would need information about its internal structure, along with the power consumption profile for the complete instruction set architecture (ISA), depending on the profiling methodology we will choose. These pieces of information are practically always kept secret by the manufacturer, partly because disclosing them would compromise the secrecy about their intellectual property. Therefore, we need to extract the model from the actual manufactured chip.

In this work we focus our analysis only on the first part of this tool, that is, the building of the oracle. Integrating such a tool in a compiler driver is a possible future work.

1.3. Thesis structure

In Chapter 2 we present the state of the art with respect to the energy and power estimation field; then, in Chapter 3 we will describe the tool we built, and its structure; Chapter 4 we illustrate the model performance indicators, and we show the model results with some test benchmarks; lastly, in Chapter 5 we discuss the feasibility of the more general oracle development process.

2 | State of the art

Estimating the power consumption of a computing device is a relatively researched topic. There have been different approaches, depending on the technique chosen. Ideally, we are interested in devising a perfect oracle, able to receive as inputs the same inputs given to the real device, and to give as output the accurate estimate of the power dissipated by the device, or the energy consumed in the process. The two may or not be equivalent, since for different use cases the device could perform a terminating calculation, or run a service that must be always available. In the former case one is probably interested in the energy required in the process, in the latter the power (i.e., power per unit of time) may be more useful. In this work we studied an embedded device, usually employed to run a service, so in the following we will restrict our analysis on the power consumption.

2.1. Estimation techniques

The distinction on how do we build the oracle (and on how does it work) shows a large range of granularity. A first distinction we can do is about the general principle at the base of the oracle working. Such a distinction can be very broad, and possibly encompass very different techniques. We briefly discuss which are the best for our analysis.

A first possibility is to perform a *simulation* of the processor under test, i.e. to simulate the hardware components and their behaviour (including the power consumption) with a software program [9]. The simulator receives as input a description of the device in a suitable hardware description language (HDL), possibly (but not necessarily) Verilog [11] or VHDL [12], along with some code to execute on its physical counterpart, and it simulates the processor working (up to the simplified physical effects). In this way, we can obtain an estimate of how much energy the computation will consume. This process is slow, but more importantly it requires a physical modelling of the processor, based on accurate information of its internal workings and architecture. This information is rarely disclosed by the manufacturer, so some approximations and speculations may be made, for instance by providing higher level descriptions than the HDLs. The goodness of the model result is as good as the description accuracy.

Another approach is to rely on tools provided directly by the manufacturers, such as hardware performance counters (HPCs) and performance monitor counters (PMCs) [9]. Such tools are provided by the CPU, accessed by specific registers or files, and they measure appropriate metrics about the CPU workings (like elapsed clock cycles or directly a power measurement). Their usage is very straightforward, but their accuracy greatly varies [7], with errors up to 32 %, so they are not a very sound approach. Moreover, they require running the code on the physical processor, which is a setup that may be unfeasible when developing an application. For instance, the application may involve performing intensive computation, or connecting to network resources. This is a cumbersome but necessary workflow when developing the application logic: if the process is repeated for every compiler optimization pass, it may scale poorly.

Using a multimeter to perform *direct measuring* on the CPU is clearly the method that by definition provides the "perfect" measure, but it requires deploying the program, run it on the processor, and measure the current with a potentially complex apparatus. It is unlikely that such an ecosystem is at disposal of software developers, along with all the problems highlighted with HPCs. However, we can use direct measures to build a *ISA energy model*, i.e. a model that extract a power estimate from just the binary code that is going to run on the processor. Such models are generally constructed from a large dataset of measures taken on the physical processor, but after they have been properly tuned, they can provide a quick and ideally accurate prediction, without the need to access to the training data or to run the profiled code. This approach has been intensively explored, and a number of models have been tested, with different results (as seen in the following).

When comparing these techniques together, as shown in Table 2.1, ISA models offer the highest easiness of use for the developer, require no external setup when doing the prediction, and provide sufficiently accurate results. For this reason we concentrate our analysis on these models in this work.

2.2. Previous works on ISA models

2.2.1. General hypotheses

Numerous approaches have been proposed on the topic. In the following, we present a few relevant works about power and energy estimation. While some of the proposals are quite different from one another, there are some general hypotheses we can abstract from the analysed works. We can analyse our survey results along two parallel but distinct dimensions: how much information do we need to feed to the model (i.e., the model

Table 2.1: Summary comparison of the power estimation techniques.

Approach	Complexity	Required knowledge	Easiness of use by developers	Accuracy
Simulation	medium	very high	medium	good
HPCs	low	little	medium	low
Direct measuring	medium-high	low	very low	perfect
ISA energy model	high	variable	high	variable

complexity), and how much *a priori* information is needed to perform the analysis.

An ideal oracle would analyse the code that is going to be run on the processor, and extract some information, characteristics and statistics from it. Those data can generally be modelled as a series of variables $V = x_1, x_2, \ldots, x_n$. Reasonably, we expect the model to be as accurate as many variables we have regarding the input. For example, one may consider the number of instructions to complete the task, how often is the task performed and which is the sequence of instructions composing the task code. We present in Section 2.2.3 a work regarding the usage of similar information in the prediction. If we do not know the value of all the variables, we can try to recover manually (when it is possible) the missing ones from the others or from the inputs. As a simple example, if we do not know the number of clock cycles taken by the task, but we have the sequence of instructions and the information on how many cycles each instruction takes to execute, we may estimate the total number of clock cycles. In the end, we substitute at each missing variable x_i a term consisting in a function of other variables or external data: $f_i(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n, e_1, \ldots, e_m)$, where e represent externally available data (such as a table containing the information on the clock cycle count).

Even when we cannot recover the missing variables with this approach, we may be able nonetheless to estimate them by assigning an heuristic value. For example, if we do not have the number of clock cycles each instruction takes, we can compare our model with other microprocessor architecture, and port the data to the ISA under test. In such a case, the underlying assumption would be that the two CPUs will have different but similar cycle counts: for example, it is a quite safe assumption to say that performing a division takes way more time than performing an addition, in every commercially available microprocessor.

More interestingly, there are some variables that cannot be extracted from available information, but that can be inferred as the results of some experiments from the processor. In particular, a lot of parameters characterizing the model are unknown *a priori*, but can be extracted by fitting the model for them. This is a large and widely used ecosystem of techniques (of which the linear regression is one of the most used [4]), that yields statistical correct results. An example can be the base power dissipated by the processor at idle, when no calculation is taking place, or the cost of flipping a single bit in a certain data path.

Depending on the variables that we are trying to find, an approach can be more effective or efficient than another, as the examples we have made suggest. In researching the related works on the topic, we can start from these observations to explore the current state of the art on these approaches, evaluating how much information is needed for each one.

2.2.2. Ideal power estimation

A common starting point when studying power consumption is [26]. There, the authors outline an idealized model for the energy consumed by a program:

$$E_P = \sum_i (B_i \cdot N_i) + \sum_{i,j} (O_{i,j} \cdot N_{i,j}) + \sum_k E_k \tag{2.2.1}$$

where

- E_P is the total energy cost of the program.
- for each instruction i, B_i is the base cost of i, that is a constant cost attributed at each instruction, and N_i the number of times i gets executed.
- for each couple of consecutive instructions i and j, $O_{i,j}$ is the circuit state overhead, a term that summarize all the possible dispersions due to circuit switches, capacitances, and in general the switching from one instruction to another. $N_{i,j}$ is the number of times that particular interleaving of instructions occurs.
- E_k is the energy consumed due to other inter instructions effects k. Examples of this are cache misses and stalls in the CPU pipeline.

The usage of the term $O_{i,j}$ is important, because it accounts for all the charge-discharge cycles of the parasitic capacitances inside the microprocessor.

Note, however, that this model estimates the energy required by the program: since we are studying the power consumption, we need to slightly refine it, by dividing by the

total time elapsed. In doing this, the first term of the sum may be simplified: the energy cost for each instruction can be expressed as $B_i = B_i^P c_i T_c$, that is the product of the power consumption, the number of clock cycles needed by the instruction, and the clock period; similarly, the total elapsed time is $T = c \cdot T_c$, the product of the total number of clock cycles elapsed and the clock period; finally, the total number of clock cycles can be expressed as the sum of all the clock cycles used by all the instructions, that is $c = \sum_i c_i N_i$. Overall, the first term of the sum in Equation (2.2.1) can be expressed as

$$\frac{\sum_i B_i \cdot N_i}{T} = \frac{\sum_i B_i^P c_i T_c \cdot N_i}{c \cdot T_c} = \frac{\sum_i B_i^P c_i T_c \cdot N_i}{\sum_i (c_i N_i) \cdot T_c} = \frac{\sum_i B_i^P \cdot c_i N_i}{\sum_i c_i N_i} \tag{2.2.2}$$

that is, a weighted mean where the clock cycles act as weights. The other terms express costs related to the number of times a certain event happens during execution, so it is difficult to express them as power consumptions, without deeply changing the formula. We can simply compute them as the total energy over the total time, as in Equation (2.2.3), or we can integrate them into the other terms. The latter solution has been employed in [8], discussed in Section 2.2.5.

$$\begin{aligned} P_P = \frac{E_P}{T} &= \frac{\sum_i (B_i \cdot N_i) + \sum_{i,j} (O_{i,j} \cdot N_{i,j}) + \sum_k E_k}{T} \\ &= \frac{\sum_i B_i^P \cdot c_i N_i}{\sum_i c_i N_i} + \sum_{i,j} \frac{O_{i,j} \cdot N_{i,j}}{T} + \sum_k \frac{E_k}{T} \end{aligned} \tag{2.2.3}$$

This model is theoretically extremely precise, but practically useless on its own, since to use it we need a way to accurately estimate all the parameters. This is the major difficulty of the experimental analysis. In this model, for instance, the terms E_k^P may be particularly variable. In an ad-hoc created experiment, when all the parameters are known (or reasonably averaged), the authors obtained an error of 0.26 %; in a less ideal setting, the error rose to around 3 %. The authors also highlight some phenomena observed from data: for instance, they note that power consumption is generally quite dependent from the register allocation in the executed code. This suggests that computing these determining factors may improve the accuracy of the model, which is the approach of Section 2.2.3.

2.2.3. Using functions of the code

To better expand the parameters used in the model equation 2.2.1, Lee et al. in [15] suppose that the power consumption can be estimated by a linear model. In particular, they assume the system under test is a black-box, and try to infer its behaviour by extracting data in a stimulus-response approach. This does not mean that no knowledge about the system is required: the model the authors suggest is dependent on some information about the code (and in lesser part, the processor). This however is still an improvement, as we shifted the source of knowledge from the processor (about which we usually know very little) to the code it executes (which is on the other hand completely knowable, and even controllable to an extent with compiler optimizations).

They focus their model on the processor pipeline. We can suppose the total energy E required by a program is the sum of all the energies consumed in the various clock cycles E_i. This term can be calculated as the sum of the energies consumed by all the stages of the processor pipeline:

$$E_i = \sum_{s \in S} e_s(I_s(i), I_s(i-1)) \tag{2.2.4}$$

where $I_s(i)$ is the instruction occupying pipeline stage s at clock cycle i, and $e_s(X,Y)$ is a function representing the power consumption of instruction X, when it is following instruction Y. This model suggests the importance of analysing the instruction sequence as couples of instructions, not single ones. This emphasizes the effect of the circuit switching cost $O_{i,j}$ in Section 2.2.2.

To define the function $e_s(\cdot,\cdot)$, we identify some known properties of the processor, from which we extract the regression variables that constitute the core of the model [4]. As an example, we may assume that the processor dissipate energy every time a logic gate is activated (this is consistent with the modelling that couples every complementary MOS (CMOS) gate with a parasitic capacitance). Then, we deduce that every bit asserted in the binary instruction will activate some areas of the processor, thus increasing the energy cost. As a result, we hypothesize to use as a predictor variable the binary weight of the binary instruction, that is the number of 1s in the binary representation of the instruction. This turns out to be a correct guess, so precise that it is possible to extract sensitive information from the power absorbed by a processor [19].

The authors list a few possible variables:

- the *instruction fetch address* of the instruction. It may reflect the bus power consumption and the cache hit rate.

- the *instruction bit encoding* of the instruction. It affects the power consumption related to the instruction register.
- some *operand specifiers*, like the register numbers or the immediate operand of instructions. They have similar effects to the instruction bit encoding.
- the *program variable values*, which affect the state of the arithmetic-logical unit (ALU) of the processor, and thus ultimately their consumption

Once defined the set V of model variables, the function $e(\cdot, \cdot)$ is defined as:

$$e_s(X, Y) = B_s^X + \sum_{v \in V} f_s^X(v_X, v_Y) \tag{2.2.5}$$

where B_s^X is the base cost for instruction X in pipeline stage s, v_i is the variable v calculated for instruction i, and f_s^X is a suitable function expressing the power cost deriving from the variables. The authors suggest the function

$$f_s^X(v_X, v_Y) = H_s^{v/X} \cdot h(v_X, v_Y) + W_s^{v/X} \cdot w(v_X) \tag{2.2.6}$$

where $h(i, j)$ is the Hamming distance between numbers i and j expressed in binary, $w(i)$ is the binary weight of number i, while $H_s^{v/X}$ and $W_s^{v/X}$ are parameters to be fitted through linear regression (specifically, the least square method).

This model is interesting for two reasons. First, it allows, in principle, to explore arbitrary values of the variables v via ad-hoc written programs. Second, linear regression and the ordinary least square (OLS) method are widely accepted tools, that benefit from a solid set of analysis we can perform from it, like testing the validity of the used predictors using hypothesis testing.

The results presented by the authors are very promising: the R^2 coefficient of determination [4] of the resulting model is 0.9893, and the average relative error is around 2.5 %, with a maximum error of 6.33 %.

2.2.4. The Dual Bit Type method

A completely different approach is proposed in [13], where the authors try to model a power consumption score in a purely statistical approach.

The authors adopt the usual modelling for a digital logic gate, whose power consumption is $P = CV^2 f$, where C is the inevitable parasitic capacitance associated with the gate, f is the frequency of the clock operating on the gate, and V is the voltage applied to its ends.

This model holds when the gate is constantly switching its state. Since this realistically never happens, the formula needs to be refined by adding a statistical correction factor. The power of each gate is then calculated as $P(0 \to 1) \cdot CV^2 f$, where $P(0 \to 1)$ is the 0 to 1 transition probability of the signal.

It is possible to adapt this concept to more complex architectural units. For instance, capacitance of components like adders, subtracters, multiplexers, barrel shifters and registers scale linearly with the number of bits of the component:

$$C_T = C_{eff} \cdot N \tag{2.2.7}$$

where C_{eff} is the capacitive coefficient (ideally, the capacity switched for each bit in the module), and N the number of bits of the module.

Other components behave in a more complex manner. For instance, an estimate of the power of an array multiplier may be $P = CN^2V^2f$, where N is the dimension of the inputs (again, a statistical correction is needed). Note the quadratic dependence on the input bits; this however is not a problem for this analysis. Or, the component may be described using several different capacitance terms: in such a case, we also need to use several coefficients in (2.2.7). In practice, we can resort to a vector model:

$$\boldsymbol{C}_T = \boldsymbol{C}_{eff} \cdot \boldsymbol{N} \tag{2.2.8}$$

where

$$\boldsymbol{C}_{eff} = \begin{bmatrix} C_0 & C_1 & \cdots & C_n \end{bmatrix} \qquad \boldsymbol{N} = \begin{bmatrix} N_0 \\ N_1 \\ \vdots \\ N_n \end{bmatrix} \tag{2.2.9}$$

For each subpart the component is made of, C_i indicates the capacitance of the subpart, N_i the variable the corresponding capacitance is dependent on, and n is the number of subparts. This way it is possible to manually tune the different number of subparts of the model (provided we are able to identify them).

In theory, all the models for the various components can later be fitted to find the optimal value for each C.

As we said, it is a widely accepted assumption that the power consumption of a processor is also determined by the inputs applied [19]. This suggests that in order to have a precise

power estimation we should take into consideration the input applied to the system. Opting for the statistical point of view of the analysis, we can shift our focus to the statistical distribution of such input. A naive approach to this can be assuming the input will be distributed as a uniform white noise (UWN). This is extremely imprecise, and leads in corner cases to errors up to 6543 % [13].

This estimate is improved by the authors by noting that, in general, input to a system is never a true UWN, but instead the most significant bits (MSBs) have lower switching probabilities than the least significant bits (LSBs). The key proposed correction, called the dual bit type method, analyse the switching activity of the input stream, and assign different capacitances to different switching probabilities.

Then, to analyse the different behaviour in the input data, we can observe the regions of bits that have different statistical distributions, and identify their boundaries (called breakpoints). This analysis can be simple or complex, but as a first approximation the authors suggest taking as breakpoints the values $BP1$ and $BP0$, obtained as

$$BP1 = \log_2 (|\mu| + 3\sigma) \tag{2.2.10}$$

$$BP0 = \log_2 \sigma + \Delta BP0 \tag{2.2.11}$$

$$\Delta BP0 = \log_2 (\sqrt{1 - \rho^2} + |\rho|/8) \tag{2.2.12}$$

where μ, σ and ρ have the usual statistical meanings of mean, variance and correlation, referred to the input distribution.

Another consideration can be that the MSB of a signal is often a sign bit: this switches with particular rules, and we are able to improve the model for it. Comprehensively, to account for the different data types, we use one capacitance coefficient for each type of data modelled. The uniform part of the signal (UU) may be represented with one coefficient, since it is well described by a classic, UWN model. The sign part (SS), on the other hand, exhibits four possible transitions, therefore it may be modelled using four different capacitance coefficients (C_{++}, C_{+-}, C_{-+}, C_{--}).

A last improvement the authors make is to model the various functions a single component (like an ALU) can do by using different sets of coefficients. This in practice partitions the input bits from another point of view, in operand bits and control bits.

All these coefficients can be found by fitting the model against a proper input dataset. To generate that, the authors use an automatic pattern generation tool to produce inputs that stimulate all possible data capacitance of the model. Finally, the least square regression can be used to find the coefficients.

The results are very good in the sense they have low error rates. However, correctly modelling the various components requires an in depth knowledge of the internals of the processor (this method has been described as “white box approach” [5] and “grey box model” [21]).

2.2.5. Precise multi-threading processor estimation

If we have access to more information about a processor, we can further refine our model, and craft an ad-hoc one that leverages the processor-specific information we have. Georgiou et al. in [8] use a XMOS XS1-L “Xcore” processor, about which the authors know the thread behaviour and other information. Specifically, the Xcore is a “time deterministic multi-threaded architecture”. This means the authors know that the processor has a four stage pipeline for all instructions, but also that the stalls due to the fetch of new instructions can be known in advance [17]. The used model is:

$$E_{prg} = (P_s + P_{di}) \cdot T_{idl} + \sum_{i \in prg} \left(\frac{P_s + P_i M_{N_p} O}{N_p} \cdot 4 \cdot T_{clk} \right) \tag{2.2.13}$$

where

$$N_p = \min(N_t, 4) \tag{2.2.14}$$

and

- E_{prg} is the energy consumed by the program
- P_s is the static power of the processor (consumed at idle)
- P_{di} is the dynamic idle power
- T_{idl} is the idle time
- P_i is the dynamic power for each instruction
- O is a constant inter-instruction overhead, much similar to the circuit state switching overhead $O_{i,j}$ of Section 2.2.2
- M_{N_p} is a scaling factor accounting for the number of threads in the pipeline
- N_p is the number of instructions in the pipeline
- N_t is the number of active threads
- T_{clk} is the clock period

Again, this model predicts the energy cost, but it can be modified to instead model power

consumption:

$$P_{prg} = (P_s + P_{di}) + \sum_{i \in prg} \left(\frac{P_s + P_i M_{N_p} O}{N_p} \right) \tag{2.2.15}$$

The actual calculation of the power is slightly more delicate, and we will discuss its validity in Section 3.1.3.

The results are mixed: on one hand, they are overall very promising, as the average absolute error is about 3.2 %. Moreover, the analysis done in the work are relevant both at the ISA and the LLVM intermediate representation (LLVM-IR) level, and efficiently predicts the energy savings due to code transformations. This can potentially enable a cost-driven compiler optimization framework. On the other hand, some benchmarks performed poorly, with error peaks of $\approx 10\,\%$.

Moreover, we must consider that this model does assumptions that may not be easily adaptable to other processors. For instance, the inter-instruction overhead O is taken as constant, which may not necessarily be the best approximation in all cases (although it is scaled with respect to the number of instruction in the pipeline). Or, the dynamic power of the instruction may depend on some characteristic of the code, as explained in Section 2.2.3, more in some processors than in others: therefore not considering it in any model may lead to errors for those devices that do show a dependency.

However, this approach achieved an high precision, and requires relatively little knowledge about the processor internals compared to other models (like the model of equation (2.2.1) or the one described in Section 2.2.4). We can therefore take inspiration from the power model of equation (2.2.15), and adapt it to another processor with minor modifications.

2.3. Conclusions

Our ideal analysis of a new processor should

1. be processor-agnostic, meaning it could be theoretically applied to any general purpose processor
2. not require any knowledge about the processor internals (or at least require as little as possible)
3. fast and easy to be conducted, meaning new processors can be tuned in a relatively fast cycle

None of these requirements in necessary to have a good estimate: however, their compliance would allow the analysis to be realistically integrated into a widely used compiler

development. The only practical requirement in such a case would be having access to a physical example of the processor to be tested, and to a suitable measuring apparatus (we describe ours in Sections 3.1.2 and 3.1.7). Requirements 1 and 2 allow the analysis of a new processor to be straightforwardly conducted on the device, without having to refine the model in an ad-hoc manner each time; requirement 3 allows such analyses to keep up with the processor production and the development of compilers.

The approach of Section 2.2.2 is simple, but we need a way to estimate the parameters. In any case, it provides a good general approach to what our model should be. To actually conduct the estimate, we can take inspiration from different ideas from other approaches. From the approach of Section 2.2.3 we can take the usage of arbitrary functions of the code to provide more insight about the input we are using in the experiments, thus providing more variables to perform the regression upon. From Section 2.2.5 we adopt the usage of a multiplicative inter instruction overhead factor.

The dual bit type method of Section 2.2.4 requires a slightly deeper analysis. While its results provided very low error, it must be noted that the experiments took advantage of some knowledge about the internals of the device under test (DUT). This does not necessarily break requirement 2, but we can expect the precision of the method would decrease if applied to a general "one size fits all" processor model. A possible solution worth mentioning is that we can differentiate our model to provide a small number of model classes. In that way a new processor can be fitted using all the possible models, and then the best performing one can be chosen (we can assume its goodness is a sign of closeness of the model to the reality). However, we preferred to proceed with a single model, leaving this possibility as a future work.

Moreover, the dual bit type analysis is more complicated than plain OLS, as it requires doing an ad-hoc part of model construction based on the input structure. This partially breaks requirement 3 and 1. However, we can adopt from it various insights about the structure we can highlight in the input code: for instance, we can expect that various components of the processor will contribute with different amounts to the total, predict what these components are, and detect their usage using the functions of Section 2.2.3.

In Table 2.2 we show a summary of the analysed works. Since the models (2.2.4) and (2.2.15) are those requiring the littlest knowledge about the processor internals, we chose them as starting point for our model. It is worth noting that while they provide good results, they also provide a quite high experimental difficulty, which may make the experiments difficult and yield to suboptimal results.

Table 2.2: Summary comparison of the analysed previous works.

Previous work	Section	Complexity	Required knowledge	Experimental difficulty	Results
Basic	2.2.2	low	high	high	good
Functions of the code	2.2.3	medium	little	medium-high	very good
Dual Bit Type	2.2.4	high	high	medium	very good
Multi-threading model	2.2.5	medium	little-medium	medium-high	very good

3 | Power model construction

As noted in Section 2.3, we need now to design a power model starting from the work of [8]. Our analysis was feedback driven, in the sense that while taking measurements, the insights observed in the measures are translated to improvements to be applied to the model. We show the major components of our oracle.

3.1. Experimental components

3.1.1. The processor

Since our analysis aims at devising a profiling method applicable to the most large set of processors (ideally, all of them), the first step is to choose a processor to use as example, that is, our DUT. We resorted to the STMicroelectronics™ microcontroller unit (MCU) *STM32F407VGT6*.[a] It is part of the ARM® Cortex®-M4 processor family, is a 32 bit architecture, runs the Armv7E-M ISA, meaning it supports Thumb instructions, has an hardware floating point unit (FPU) [25], and is easily programmable by end users using STMicroelectronics' tools. The MCU is hosted on a STM32 Discovery kit board.[b]

3.1.2. The measurement tool

In principle, to measure the power flowing into the DUT we need a way to measure both the voltage and the current between two points of a circuit. The former can be relaxed by observing that a digital processor usually works under a very specific voltage (usually 3.3 V or 5 V), however, knowing the exact tension drop leads to more accurate measurements, as we can keep track of voltage fluctuations in the circuit. We used the Qoitech™ Otii Arc Pro, an easy to use digital multimeter capable of measuring time varying tension with a decent precision, as showed in Table 3.1. Since the device actually measures only tension, the current reading is obtained by using a shunt resistor, as explained in Section 3.1.7. The Otii Arc Pro comes with a graphical user interface (GUI) program delivered by the

[a] see `https://www.st.com/en/microcontrollers-microprocessors/stm32f407vg.html`
[b] see `https://www.st.com/en/evaluation-tools/stm32f4discovery.html`

Table 3.1: Precision of the physical quantities measured by the Otii Arc Pro.

Physical quantity	Precision
Time (s)	1 ms
Current (A)	0.1 µA
Voltage (V)	1 mV

manufacturer, from which it is possible to start and stop a recording while monitoring the plot of the physical quantities of interest on the screen. While recording, the Otii Arc Pro measures the tension at its probes with a sampling frequency of 1 kHz. From the GUI, it is then possible to export such data in a CSV file.

As shown in Section 3.1.5, to produce an accurate model a huge training dataset is required. The Otii Arc Pro allows to take measurements only using the GUI: this can be a problem as the time to save a measurement using it will considerably be slower than using a command line tool. The straightforward solution is to use a software that allows this kind of measurements. The Otii GUI offer such an extension, via an extra commercial product. However, to mitigate this disadvantage without resorting to a commercial solution, we chose to automate the GUI measurement process using a graphical automation tool. Since the machine used for this work was running the X11 protocol, we picked the `xdotool` software,[c] which is the de-facto standard for this protocol. The script simply replays all the steps that would normally need to be performed manually to measure and save the power measure. To get a better reading, we recorded each measure for 3 s, to then obtain an average value. The total measurement process takes around 20 s.

3.1.3. The tentative mathematical model

The data we collect with the measurement apparatus are meant to be used to train a mathematical model. From the conclusions of Section 2.3 we can take inspiration from the equation (2.2.15), here reported for convenience:

$$P_{prg} = (P_s + P_{di}) + \sum_{i \in prg} \left(\frac{P_s + P_i M_{N_p} O}{N_p} \right) \tag{2.2.15 recall}$$

However, as we highlighted in Section 2.2.3, we also want to benefit from the linear regression theory that can be applied to such a model. In particular, we can suppose that our model can be represented by a linear model, where we can predict the power consumed

[c] see `https://github.com/jordansissel/xdotool`

by executing a particular instruction i by multiplying some variables about the instruction with some (now unknown) coefficients. For instance, to predict a particular instruction power, we can observe that

$$P = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon \tag{3.1.1}$$

where P is the power consumed by the instruction, $\{\beta_1, \beta_2, \cdots, \beta_n\}$ are the unknown parameters of the regression, β_0 is the constant cost associated with the instruction, and ϵ is the error term. Then, our prediction will be

$$\begin{aligned} \hat{P} &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n \\ &= P - \epsilon \\ &\approx P \end{aligned} \tag{3.1.2}$$

However, this poses the question of what variables should we consider. For instance, following the discussion of Section 2.2.3, we can imagine considering the binary weight of an instruction as variable (x_w). The model then requires a coefficient to be multiplied to it (β_w), representing the amount of power dissipated for each unary increment of the binary weight for the input instruction. The problem is that, in general, we have no reason to believe this coefficient is equal for all the instruction of the ISA. Therefore, we need a way to express this variability in the model, that is, we need a way to differentiate which parameter to use.

This can be addressed by considering the instruction i as a variable, and introducing an indicatory (or "dummy") variable for each instruction [4]:

$$I_i = \begin{cases} 1 & \text{if the current instruction is } i \\ 0 & \text{otherwise} \end{cases} \tag{3.1.3}$$

Then, the model becomes

$$\begin{aligned} \hat{P}_i =& I_1 \cdot (\beta_{10} + \beta_{11} x_{11} + \beta_{12} x_{12} + \cdots + \beta_{1n} x_{1n}) \quad + \\ & I_2 \cdot (\beta_{20} + \beta_{21} x_{21} + \beta_{22} x_{22} + \cdots + \beta_{2n} x_{2n}) \quad + \\ & \vdots \\ & I_m \cdot (\beta_{m0} + \beta_{m1} x_{m1} + \beta_{m2} x_{m2} + \cdots + \beta_{mn} x_{mn}) \end{aligned} \tag{3.1.4}$$

Another equivalent description of this is that we can identify one equation for every

instruction i of our ISA:

$$\hat{P}_i = \beta_{i0} + \beta_{i1}\, x_{i1} + \beta_{i2}\, x_{i2} + \cdots + \beta_{in}\, x_{in} \tag{3.1.5}$$

so that in the end we have

$$\hat{P}_i = \begin{cases} \beta_{10} + \beta_{11}\, x_{11} + \beta_{12}\, x_{12} + \cdots + \beta_{1n}\, x_{1n} & \text{if instruction is n°1} \\ \beta_{20} + \beta_{21}\, x_{21} + \beta_{22}\, x_{22} + \cdots + \beta_{2n}\, x_{2n} & \text{if instruction is n°2} \\ \beta_{30} + \beta_{31}\, x_{31} + \beta_{32}\, x_{32} + \cdots + \beta_{3n}\, x_{3n} & \text{if instruction is n°3} \\ \vdots \end{cases} \tag{3.1.6}$$

From the point of view of linear regression, this is an example of the usage of a qualitative variable (the current instruction) in the linear model, accounting for the interaction effect of the qualitative variable with the quantitative ones. Note that in the end some variables gets multiplied together. For instance, expanding the first term of equation (3.1.4) we get

$$\beta_{10} \cdot I_1 + \beta_{11} \cdot I_1\, x_{11} + \beta_{12} \cdot I_1\, x_{12} + \cdots + \beta_{1n} \cdot I_1\, x_{1n} \tag{3.1.7}$$

This however is not a problem, as to perform linear regression we only need the model to be linear in the coefficients β_{ij}, not in the variables, which can even be replaced by arbitrarily complex functions. The usage of qualitative variables is such a common situation that many data analysis software packages include a way to automatically deal with them, without manipulating manually the variables.

3.1.4. The measurement workflow

The entire measurement workflow is illustrated in Figure 3.1.1. A rectangle represents an artifact (a source file or the physical board), while an oval represents a program (whether a binary one or a script). Since we collected the measurements, wrote the data analysis code, and redacted the work using the GNU Emacs editor [22], we wrote the main orchestrating scripts in Emacs Lisp [16].

The overall process is handled by the Emacs Lisp driver, that invokes the other external programs, script and functions. The code to be flashed onto the STM32 board is obtained from the official STMicroelectronics editor, which provides a firmware and a C template. This in turn contains a `main` function with an infinite loop, and offers various injection points in which add the user code. This is consistent with our idea of measuring the power consumption (expecting an embedded device to run indefinitely), and allows us to measure

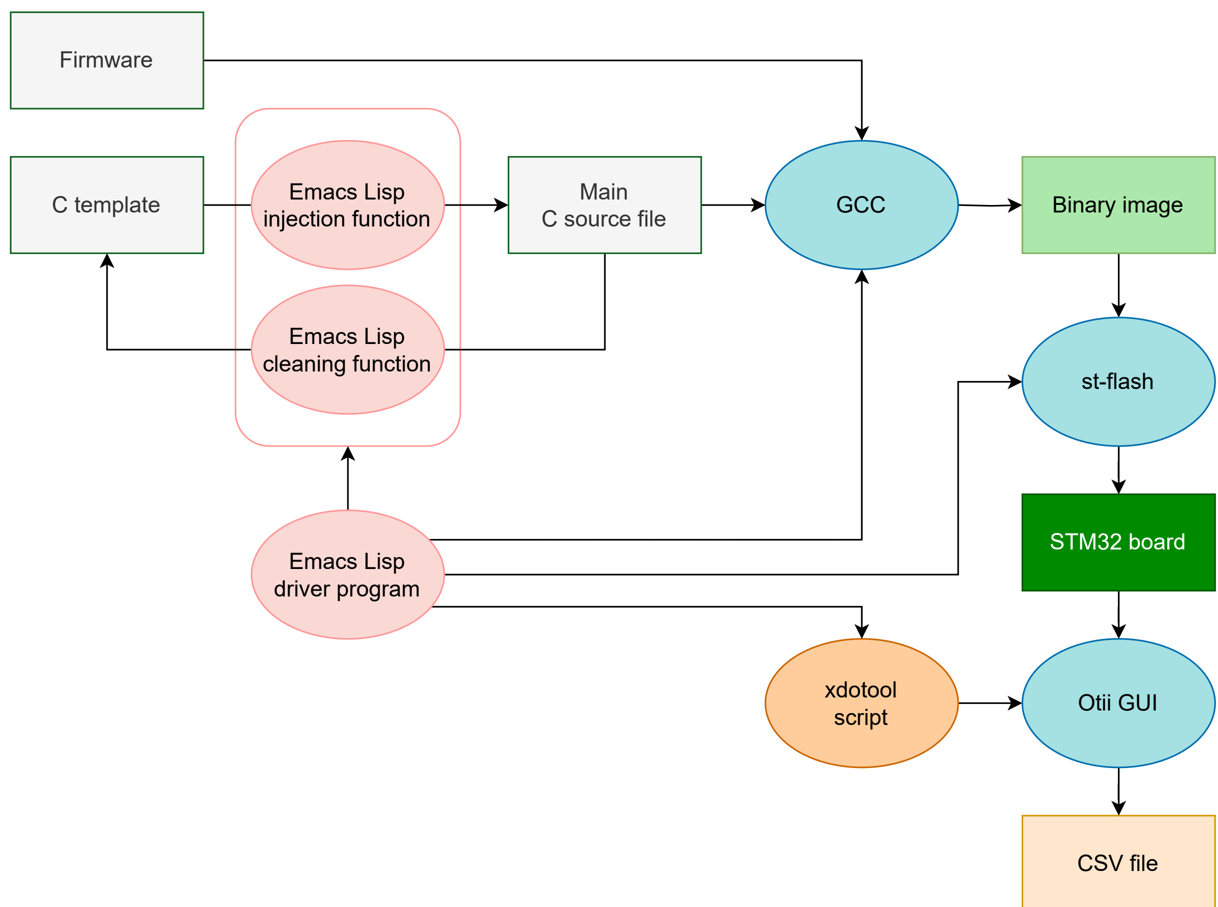


Figure 3.1.1: Measurement workflow.

the single instructions. The provided C template is modified by an Emacs Lisp function and the instruction to be tested is injected in the appropriate place (such instruction is generated with the process described in Section 3.1.5). The resulting C file is compiled and linked with the STMicroelectronics firmware, composed of C and assembly files. In this work we used GCC [23], as it is free software and it is distributed in numerous toolchains, one of which specific to compile ARM code to bare metal. The compiler outputs a binary image, that can be loaded onto the board with the `st-flash` program (also provided by STMicroelectronics). To do so, the board is connected to a computer using a USB cable. Then, the `xdotool` script is executed, which in turn drives the Otii GUI in taking the measurement. The result is a CSV file, whose name is a mangled version of the instruction just measured. Then, another Emacs Lisp function cleans the C main file to restore it to the previous state, so that the following instruction can be injected, and the process is repeated as needed.

The net result is a driver program that generates the suite of instructions to be measured, injects the code, compiles the project, flashes it onto the board, measures the power consumption, and then repeats the process, until the suite is finished.

Since we would like to measure the power consumed by the processor due to a single instruction, and nothing else, we would like to isolate the effects of the instruction from the other operations performed by the processor. This is not entirely possible, as the firmware provided is effectively a little operating system, dealing with memory events and other overhead logic. However, we can assume the overhead computation will be performed during testing as during the normal functioning, so we can factor out its contribution, that will be absorbed into the constant term of the linear model. This is not an exact analysis, as the memory management logic executed will depend on the user code running, but we can ignore this difference to provide at least a first prototype of the power model.

To assess the impact of a single instruction, a common idea is to execute it in an infinite loop. Jumping at the start of the loop requires some overhead computation that pollutes the measure, so a solution can be to not use a single instruction executed in loop, but actually a streak of identical instructions. A too long streak, on the other hand, can trigger some memory events such as cache misses. To mitigate this effect, a rule of thumb is to produce streaks of 200 instructions [26]. Since we need to inject ARM assembly code into a C template, we used the `__asm__` directive, qualified as `volatile` to prevent the compiler to optimize it away. Moreover, we can place our injected instruction into the `.rept 200 <...> .endr` directive, to automatically repeat the instruction 200 times.

A thing worthy to be noted is the extraction of the binary encoding of the instructions.

Given a particular ARM instruction, it is theoretically always possible to recreate its binary encoding, for instance using an assembler. In this work we did so by using the GNU Assembler (`as`) [6] when we needed to retrieve an encoding of a new instruction (see Listing B.1 at page 90). However, when collecting data for the model, we can easily find the encoding for the instructions we are testing, as we can scan the binary file written to the board for any instruction repeated around 200 times. Our code reads that encoding and saves it for caching purposes.

3.1.5. The train dataset generation

In order to fit the linear model, we need data about the input-output (or variable-response) behaviour. When doing a simple linear regression, we would need a so-called *train dataset* of points (x, P), where x is the input (or explanatory) variable, and P is the power output, measured with the Otii Arc Pro. In this case however we are performing multiple linear regression, and our points are $n + 1$ dimensional, where n is the number of variables we have been able to extract from the code run on the board: $(x_1, x_2, \ldots, x_n, P)$.

The first question to ask is how many points are we going to take. Theoretically speaking, the more points we have, the better the prediction is, so we can aim to test each instruction; however, simple calculations show this is not possible. We suppose we need to profile the `add` instruction, i.e. the instruction that adds two binary numbers and stores the result in a register. This instruction accepts three arguments: the destination register, the first source register, and a flexible second operand, that may be a register or an immediate value. The ISA exposes up to 11 general purpose registers, and the upper limit of an immediate integer depends on the mode of the ISA (Thumb or ARM), but for simplicity we can limit ourselves to 255. Moreover, the operand can be shifted with some specific commands: `asr`, `lsl`, `lsr` and `ror` all accepts as argument the number of bits to shift by, in the range $[1, 31]$; `rrx` has a single meaning.

Summing up, many instructions in the ARM ISA accept 11 possibilities as destination register, 11 possibilities as first source register, and $255 + 11$ possibilities for the second operand (again, this is subject to differences regarding the mode of the ISA, but for simplicity we are ignoring this detail as the overall calculation will not primarily depend on it). In turn, each second operand may be manipulated by $4 \cdot 31 + 1$ possibilities. In total, to completely test the `add` instruction we need to explore

$$11 \cdot 11 \cdot (255 + 11) \cdot (4 \cdot 31 + 1) = 11 \cdot 11 \cdot 266 \cdot 125 = 4023250$$

data points. Our experimental setup, in the end, required around 20 s to perform a

single measurement, so the amount of time required is $4023250 \cdot 20\,\text{s} = 80\,465\,000\,\text{s}$, roughly 2.5 years. The ARMv7E-M ISA counts (depending on what can be considered a different instruction) around 242 instructions, easily showing that a complete exploration is prohibited.

Another way to see why it is necessary to choose a subset of data points to measure is to observe that ARM instructions are encoded as 16 and 32 bit words. Not all combinations of bits are valid instructions, but we can assume as reasonable upper bounds of the testable number of instructions the values 2^{16} and 2^{32}, which again, accepting the measurement time of 20 s, lead to respectively around 15 days and 2762 years.

What we did in this work is to select a subset of each relevant feature of the ISA instructions, so to reduce the magnitude of each factor making up the total of test cases. We can, for this, choose uniformly spaced out sample values that cover up the operand space. This in theory also collects data about various functions of the code that we are not considering right now. For instance, expecting to include the binary weight of the instruction in the linear model, by choosing a skip step that is not a power of 2 we can actually produce operand values with various binary weights. In Figure 3.1.2 we can see that this uniform sampling produces good results. The littler the step, the more data we have, and the more we get a normal distributed density. The thicker lines represent step values of 5 and 10, which greatly reduce the amount of data we get (corresponding to the area under the curve), but we still explore almost all the weight space.

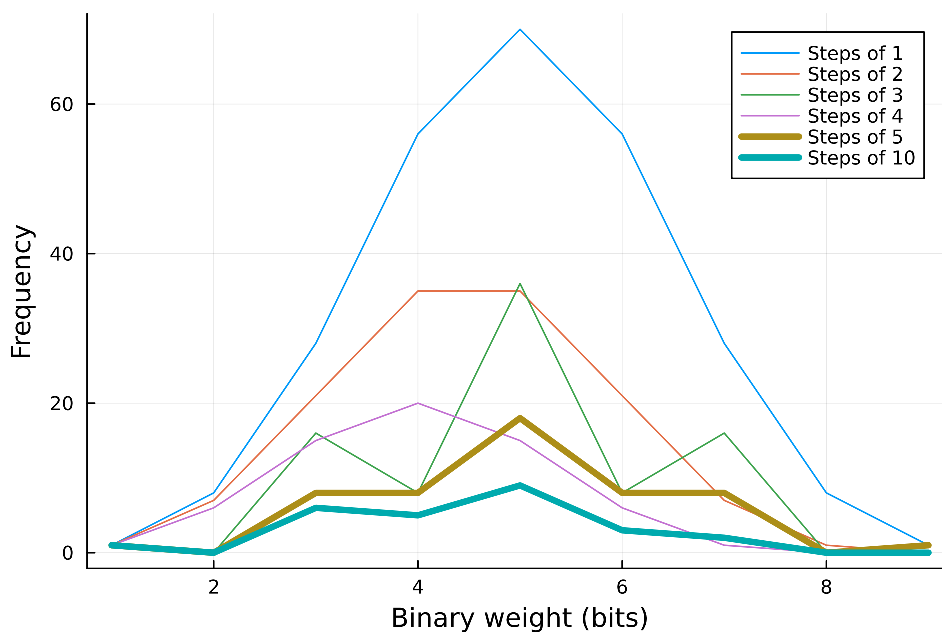


Figure 3.1.2: Binary weight densities for various operand exploration steps.

With this reasoning and sticking to the `add` example, we can then test just 3 registers for the register operand, and up to 20 values for the immediate operand. The shift feature can be reduced too, for instance by testing 6 values. Now the calculation becomes

$$3 \cdot 3 \cdot (20 + 3) \cdot (4 \cdot 6 + 1) \cdot 20\,\mathrm{s} = 9 \cdot 23 \cdot 25 \cdot 20\,\mathrm{s} \approx 29 \text{ hours}$$

This is a great improvement, but it still requires too much time.

Another way we can cut measurement time is by supposing that the shift contribution to the power consumption will be independent of the instruction considered, but instead it simply increments the power for each bit shifted. This way, the effect of the shift must be measured only once, and then the time required for each instruction reduces to

$$3 \cdot 3 \cdot (20 + 3) \cdot 20\,\mathrm{s} = 9 \cdot 23 \cdot 20\,\mathrm{s} \approx 1.5 \text{ hours}$$

We explain later how this is implemented in the actual data analysis code, however, for the purpose of the theoretical linear regression, this corresponds to modifying our model like (commas added for clarity):

$$\begin{aligned} \hat{P}_i = & I_1 \cdot (\beta_{1,0} + \beta_{1,1}\, x_{1,1} + \beta_{1,2}\, x_{1,2} + \cdots + \beta_{1,n-1}\, x_{1,n-1}) \quad + \\ & I_2 \cdot (\beta_{2,0} + \beta_{2,1}\, x_{2,1} + \beta_{2,2}\, x_{2,2} + \cdots + \beta_{2,n-1}\, x_{2,n-1}) \quad + \\ & \qquad \vdots \\ & I_m \cdot (\beta_{m,0} + \beta_{m,1}\, x_{m,1} + \beta_{m,2}\, x_{m,2} + \cdots + \beta_{m,n-1}\, x_{m,n-1}) \quad + \\ & \beta_s \cdot x_s \end{aligned} \tag{3.1.8}$$

where x_s is the amount of bits shifted, and β_s is its corresponding coefficient.

The complete code to produce the test instructions is described in Appendix A.

3.1.6. Memory instructions

Particular care needs to be taken when measuring the instructions acting on memory. The board may implement memory protection: for instance, if we execute the `pop` instruction in an infinite loop, reading from progressively changing areas of the stack, we will eventually end it and overflow in reading in prohibited regions of memory. Normally operating systems specifically prevent this; however in our analysis this is precisely the kind of behaviour we would wish for, as it would allow us to test the instruction with arbitrary arguments.

To mitigate this problem, if present, there are three possibilities. First, one could inspect

the application binary interface (ABI) of the code running, and manually write test cases that bypass the problem. Second, we could inspect the firmware to deactivate the memory protection. Third, it may be possible to bypass the firmware completely, and to inject specific binary words into the processor to be executed as instructions. This requires to know the working of the processor, and possibly using an field programmable gate array (FPGA) as a driver. However, the latter two workarounds are not guaranteed to work, as we do not know whether the board implements the protection using software or hardware components (that may not be disabled by modifying the firmware). These were anyway outside of our scope: we limited ourselves to prepare the memory by pushing onto the stack, and then executing streaks of `pop` or `push` instructions while manipulating the stack pointer.

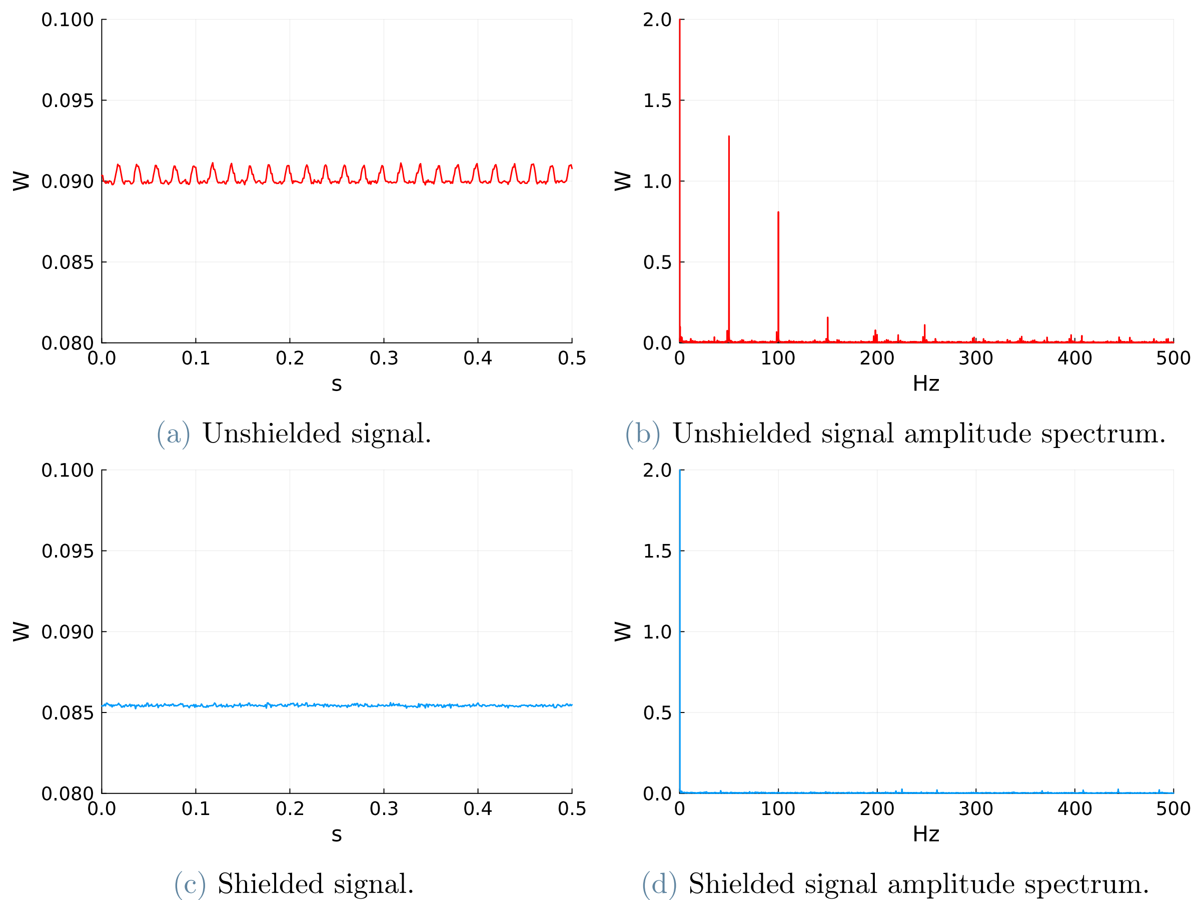
3.1.7. Electrical wiring

The STM32 board exposes two pins (C22 and I_{dd}) on the line carrying power to the processor. Normally they are short-circuited with a jumper, but it is easy to interpose a measurement apparatus directly there, measuring the current flowing in the CPU. Since the Otii Arc Pro does not directly act as an ammeter, we connected a shunt resistance between the two pins, using the corresponding contacts on the bottom side of the board. The value of such resistance is arbitrary, and can be set into the Otii GUI. This way, the program will measure the voltage drop and calculate the current flowing, using Ohm's law $V = R \cdot I \Rightarrow I = V/R$. In this work, we used three equal $(1.00 \pm 0.05)\,\Omega$ resistors connected in parallel, acting as an equivalent resistance of $0.33\,\Omega$.

Additionally, the board provides also a LED indicating communication with the PC connected to it. During normal functioning, the LED blinks at regular intervals, causing a square wave in the measured current flow. That effect can be filtered out digitally or by unsoldering the LED: here, we chose the latter method.

The shunt voltage drop is measured at the ADC pins of the Otii Arc Pro; moreover, the electrical ground of the board and the ground of the Arc (GND) have been connected.

This setup however was quite susceptible to electromagnetic disturbances, like the ones produced by the lighting fixtures, and to parasitic capacities. The current flow can be seen to sensibly change when turning on the lights in the measurement room, or when moving close an hand. To minimize this effect, we placed both the board and the Arc in a box covered in aluminium foil, which acts as a Faraday shield. Moreover, the board ground is short-circuited to the box, and that in turn was connected to the fixture electrical ground. This successfully prevented the measures from being contaminated by electromagnetic



(a) Unshielded signal.

(b) Unshielded signal amplitude spectrum.

(c) Shielded signal.

(d) Shielded signal amplitude spectrum.

Figure 3.1.3: Effect of the Faraday shield from the 50 Hz disturbance.

noise, namely the 50 Hz disturbance coming from the electrical installation. In Figure 3.1.3 we can see the two main effects of the shielding. First, the power consumption presents a little constant offset, and second, the periodic behaviour of the electric supply is practically neutralized. The spectrum of Figure 3.1.3d still presents some noise, but it is hugely lower than the unshielded one.

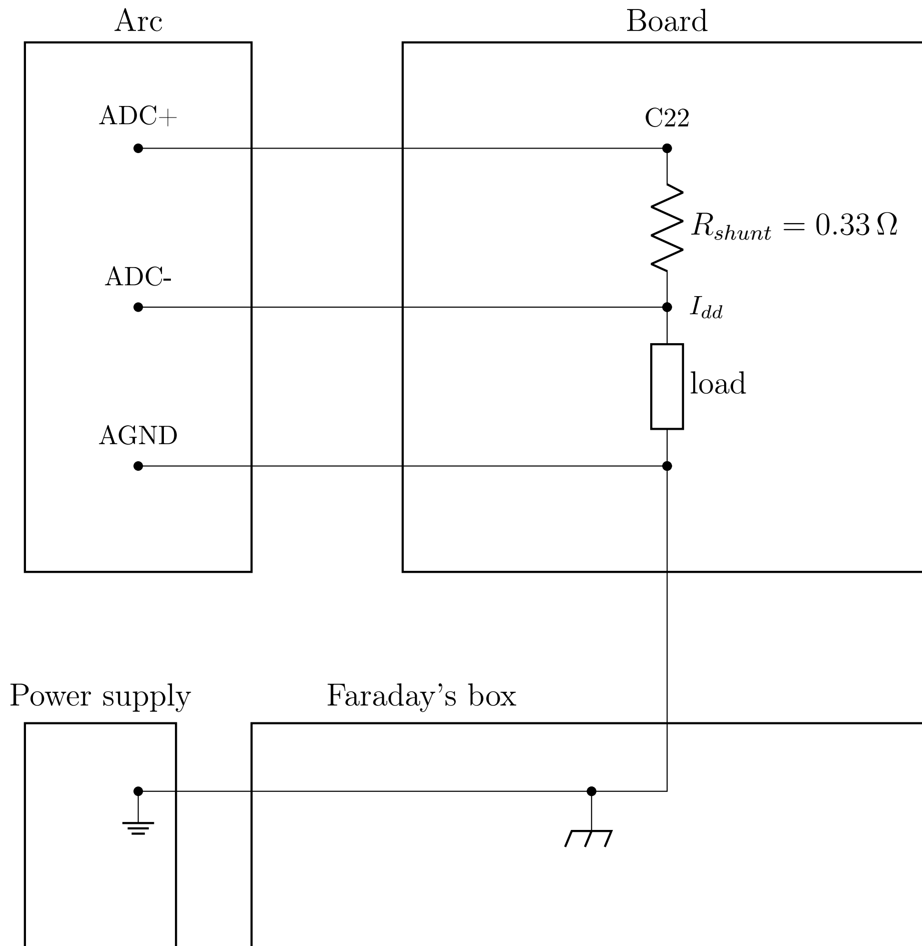


Figure 3.1.4: Circuit scheme of the measurement setup.

In Figure 3.1.4 it is shown the circuit schematic of all the electrical connection of the measurement apparatus. The generic bipole "load" is an abstraction for the CPU and all the other silicon components powered through that wire. We are only interested in the CPU consumption, but again we can imagine the overhead component power to be absorbed into the constant term of the linear regression. For simplicity, the power supplies of the various components are not shown; however, both the board and the Otii Arc Pro draw power from the computer to which they are connected via two USB cables. Such cables serve the dual purpose of transmitting energy and data, as they are used by the board and the Arc to respectively load the code in memory and send the electrical

measurements to the GUI.

In theory, the board can be powered also using an external 3 V or 5 V power source. This voltage could have been provided by the Arc; however, doing so would have meant to cut the power and connect the data USB cable between each measurement, which is time consuming and error prone, so in this work we chose to power it directly using the USB cable.

3.1.8. Data analysis

Eventually, after collecting all the data points, we need to use them to produce the linear model of the processor. In this work, we chose to process the data using the Julia programming language [2], as it is fast and easy to code in. Moreover, Julia benefits from a relatively rich ecosystem of libraries, and we used them to perform the data analysis tasks.

The first step is to collect all the data points in a unified way. We decided to use the `DataFrames` package, which provides a table-like data structure to store multidimensional data of heterogeneous types. The CSV files outputted by the Otii GUI have four fields: the readings of time, current, voltage and energy. The energy is a convenience measure computed by the program: we could use that or compute the power starting from current and tension. We chose the latter: for each CSV file, we multiplied point-wise the current and the voltage, to obtain a trace of instantaneous powers, as per the known relation $P = V \cdot I$. Then, we averaged the value to obtain a single power measure for each profiled instruction. It was easy to store the standard deviation of the instantaneous powers too: however, this turned out to be a very small value (the maximum ratio of the standard deviation to the power measure was 1.11 %), so we chose to not insert it into the successive calculations.

Next, we decorated each data point with a series of additional data, the variables of the linear regression. We extracted such variables using specific functions of the instruction under test, as suggested in Section 2.2.3. To choose which functions to compute, we combined some ideas from the analysed works with the insights shown by observing the power data plotted. This is an heuristic approach: by extracting more variables from the instructions, techniques of machine learning could notice new relationships [20]. This is left as future work. As an example, we observed that generally the instructions that have the destination register equal to one of the source register consumed a different amount of power than those who do not. A possible explanation for this is that when this happens the processor activates its internal forwarding paths, in order to not waste pipeline clock cycles:

a side effect is that different area of the processor are powered up [18]. Therefore, we decorated each instruction with a boolean value, indicating whether or not the destination register was equal to one of the source register. Clearly, this distinction makes sense only for those instructions who have a source and a destination register: as a counterexample, the instruction `pop` accepts a single register as parameter, and therefore does not exhibit this behaviour. This means that this variable has to be correlated with the instruction under test, as shown in Equation (3.1.4).

The code that constructs the DataFrame is shown in Appendix B. The characteristic of the code that will end up being regression variables are extracted using specific functions of the instruction under test. The functions are explained together with the code, while the corresponding characteristics are better explained in the data exploration phase in Section 4.1.1.

The resulting DataFrame is an 12423×14 table, upon which we can perform the regression. We chose to use the GLM package,[d] as it allows using qualitative explanatory variables, and to correlate them easily. The key point of constructing the model is to provide the formula explaining how the data relate. Listing 3.1 shows how the formula object is constructed, and how the Julia console pretty prints it. The explanatory variables are a constant term, the mnemonic opcode (this is equivalent to say that there is a uniform idle consumption, plus a base consumption for each instruction type), the usage of the "`s`" flag, the usage of conditional execution, the possible coincidence between destination and source registers, the presence of a shift operation and its amount of bits, and the presence of an immediate operand. We voluntarily left out the immediate operand amount, since in the first phases of data exploration it tended to have negative coefficients that led to significative errors in the predictions. A few variables are correlated with each other, as we expect that the impact of these characteristic of the instructions is not constant, but varies depending on which instruction mnemonic has them.

The model can be fitted using the data from the DataFrame. The result is a data structure with lots of coefficients, as the categorical variable `mnemonic` expands into a series of $n-1$ dummy variables, with n being the number of instructions profiled (we analysed 57 of them). When the model did not contain any data for a particular combination of variables, it assigns to the coefficient the value `NaN` ("not a number", according to the IEEE standard for 64 bits floating point numbers [10], which Julia uses). This means that if we later try to predict the power of an instruction of which at least one coefficient is `NaN`, the program refuses to do it, as the result would probably be meaningless. We chose to follow

[d] `https://juliastats.org/GLM.jl/stable/`

```
julia> formula = Term(mean_power_sym) ~ ConstantTerm(1) +
          (Term(:mnemonic) & ConstantTerm(1)) * (
              Term(apsr_sym) + Term(conditional_sym) + Term(dest_source_eq_sym) +
                  Term(barrel_shift_sym) * Term(has_barrel_shift_sym) +
                  Term(has_immediate_sym) + Term(binary_weight_sym)
          )
FormulaTerm
Response:
  Base power mean (W)(unknown)
Predictors:
  1
  mnemonic(unknown)
  APSR (s flag)(unknown)
  Is conditional(unknown)
  Dest reg == source reg(unknown)
  Barrel shift amount(unknown)
  Has barrel shift(unknown)
  Has immediate operand(unknown)
  Binary weight(unknown)
  Barrel shift amount(unknown) & Has barrel shift(unknown)
  mnemonic(unknown) & APSR (s flag)(unknown)
  mnemonic(unknown) & Is conditional(unknown)
  mnemonic(unknown) & Dest reg == source reg(unknown)
  mnemonic(unknown) & Barrel shift amount(unknown)
  mnemonic(unknown) & Has barrel shift(unknown)
  mnemonic(unknown) & Has immediate operand(unknown)
  mnemonic(unknown) & Binary weight(unknown)
  mnemonic(unknown) & Barrel shift amount(unknown) & Has barrel shift(unknown)
```

Listing 3.1: Linear regression model formula.

```
julia> formula = Term(mean_power_sym) ~ ConstantTerm(1) +
               (Term(:mnemonic) &
                   (ConstantTerm(1) + Term(binary_weight_sym) +
                   Term(dest_source_eq_sym))) +
               (
                   Term(apsr_sym) + Term(conditional_sym) +
                       Term(barrel_shift_sym) * Term(has_barrel_shift_sym)
                       + Term(has_immediate_sym)
               )
FormulaTerm
Response:
  Base power mean (W)(unknown)
Predictors:
  1
  mnemonic(unknown)
  APSR (s flag)(unknown)
  Is conditional(unknown)
  Barrel shift amount(unknown)
  Has barrel shift(unknown)
  Has immediate operand(unknown)
  mnemonic(unknown) & Binary weight(unknown)
  mnemonic(unknown) & Dest reg == source reg(unknown)
  Barrel shift amount(unknown) & Has barrel shift(unknown)
```

Listing 3.2: Reduced linear regression model formula.

a different approach: when the model was not trained for that kind of instruction, that means we are not sure of the precise effect of that particular combination of variables. However, we can speculate that the power can be approximated by a simpler model, for instance by not considering the effect of the correlated mnemonic. In this model, we kept some variables (like the presence of the "`s`" flag or the shift amount) uncorrelated, so that we can find a coefficient for them that is applicable to all possible instructions, even with a loss of precision. We kept the correlation with the mnemonic the binary weight and the forwarding path activation, as shown in Listing 3.2.

Even in the reduced model there are a few coefficients that are equal to `NaN`. To solve the problem, we detect those instruction that fail for both models and assign a constant power cost to them. Such constant is calculated by first averaging all the power measurement we have for each instruction mnemonic, then by averaging these mean values. In this way we produce the equivalent of a weighted sum where all the instructions weight equally. We chose to do so because some instructions are tested more than others, in a ratio that is usually not equal to the proportion in which the instructions can be observed in a typical program produced by a compiler. When measuring the instructions, some effects have been measured only for certain instructions: for instance, the shift operation of the second operand has been extensively tested for the `add` instruction, since it is one of the easiest to

write tests for, and such bias would otherwise end up in the constant computation. The exact solution would have been to test each instruction, with each combination of effects, but as we argued in Section 3.1.5, this is practically unfeasible. By assigning a uniform weight we absorb the differences due to this limitation. A future improvement is to scan the program under test by the oracle, profile the frequencies of each instruction, and compute the constant cost by that. This is still an approximation, as the exact proportion of the executed instructions is unknown until runtime, but we expect it to be a better approximation.

3.1.9. Instruction interleaving

Now we can produce an oracle that is able to receive as input a string containing an instruction, invoke on it the auxiliary functions used to decorate the data point (shown in Appendix B at page 89), and then to pass the resulting vector of values to the linear model to produce the power estimate. However, such estimate is relative to an instantaneous power of a single instruction: in a real program, instructions of different types are interleaved. To estimate the power consumption of the entire program, we can suppose to compute it by dividing the energy consumed by the program by the time the program has been run:

$$P_P = \frac{E_P}{T_P} \tag{3.1.9}$$

In turn, the energy consumed is the sum of the power consumed by each instruction, multiplied by the time the instruction has been in the processor:

$$E_P = \sum_{i=0}^{N_i} P_i \cdot T_i \tag{3.1.10}$$

where P_i is the power consumed by instruction i, T_i is the time the instruction stays in the processor pipeline, and N_i is the number of instructions making up the program. Furthermore, T_i can be seen as the product of the number of clock cycles instruction takes (c_i) and the clock period (Δt):

$$T_i = c_i \cdot \Delta t \tag{3.1.11}$$

Calling N_c the total number of clock cycles required by the program, we similarly observe that $T_P = N_c \cdot \Delta t$. We can now simplify the computation of the power of the program as

$$P_P = \frac{\sum_{i=0}^{N_i} P_i \cdot c_i \cdot \Delta t}{N_c \cdot \Delta t} = \frac{\Delta t \cdot \sum_{i=0}^{N_i} P_i \cdot c_i}{N_c \cdot \Delta t} = \frac{\sum_{i=0}^{n_c} P_i \cdot c_i}{N_c} \tag{3.1.12}$$

that is, the mean of the sequence of instruction powers, weighted with the instruction clock cycles. Note that in general the number of clock cycles for different instructions will be different; for instance, for the Cortex-M4 processor, an `add` instruction takes 1 clock cycle, but a `div` instruction takes up to 12 cycles.

These data can usually be found on the processor programmer's manual. For the Cortex-M4, both the clock cycles for the integer[e] and floating point[f] instructions are provided. Such information was adapted into a Julia data structure (specifically a Dictionary, an associative array).

In Listing C.1 of Appendix C.1 it is shown the overall function `predict_instruction`, that receives as input a DataFrame consisting of a single row, representing the decorated instruction, and outputs the tuple (P_i, c_i), i.e. the instantaneous power for that instruction and the number of clock cycles it requires.

In Equation (2.2.15) the authors introduced a constant multiplicative overhead O to model the dissipated energy due to the switching activity of the logic gates of the processor. In the cited work the authors focus themselves on the number of active threads in a given time instant, instead of the clock cycles. We have various possibilities to adapt this idea in our model. We can:

1. add a constant multiplicative factor to the final instruction power estimates
2. add a constant multiplicative factor to the instruction instantaneous power (thus, it will be later scaled by the clock cycles taken by the instruction)
3. add a constant additive factor
4. use a non-constant factor

Approach 1 in our model is equivalent to scaling the final estimate of the entire program, so it is of no use in capturing the inter-instruction behaviour. Comparing approaches 2, 3 and 4, data show that modelling the inter instruction overhead with a second linear model leads to better results. In practice, we use the pair of instruction (i_n, i_{n+1}) (decorated with the shown metadata) to build a linear regression model that predicts, for a new couple of instructions, how much will it be the inter-instruction multiplicative overhead.

---

[e]see `https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions`

[f]see `https://developer.arm.com/documentation/ddi0439/b/BEHJADED`

In the end, our model becomes

$$P_{prg} = \frac{\sum_{i \in prg}(P_i \cdot O_{i,i+1} \cdot c_i)}{\sum_{i \in prg} c_i}$$

where the values P_i (the power of the single instruction i) and $O_{i,i+1}$ (the multiplicative overhead factor between instructions i and$i+1$) are predicted with the linear models.

To collect the data for this model, we slightly adapted the code used to measure the single instructions. The resulting code allows us to call

```
(instruction-cartesian-product "add r0, r0, r%1 \\n\\t sub r0, r0, %2"
                               (list (parse-range "0:50:50" 1)
                                     (parse-range "0:10:30" 2)))
```

yielding

```
add r0, r0, r0 \n\t sub r0, r0, 0
add r0, r0, r0 \n\t sub r0, r0, 10
add r0, r0, r0 \n\t sub r0, r0, 20
add r0, r0, r0 \n\t sub r0, r0, 30
add r0, r0, r50 \n\t sub r0, r0, 0
add r0, r0, r50 \n\t sub r0, r0, 10
add r0, r0, r50 \n\t sub r0, r0, 20
add r0, r0, r50 \n\t sub r0, r0, 30
```

The injected instructions are recognized by the C compiler as separate thanks to the escape sequence "`\n`" (the "`\t`" is by convention), so they assemble to a streak of alternating instructions. The resulting CSV file is then saved in a different directory, so that Julia can use the right dataset for the two models.

Auxiliary code is added to recognize the two instructions that made up the double data point, so that the metadata functions can be called on each half separately. Such functions are the same of the single case, but the formula for the regression is modified, as shown in Listing 3.3. We chose here too to implement two models: one more specific used as default, and one more generic used as backup in case the first was not trained for the incoming instructions.

The function doing the prediction for a complete trace is shown in Listing C.2 of Appendix C.2.

```
julia> basic_formula = @formula(scale_factor ~ 1 + avg_power_clean *
           (hamming_distances + first_weight + second_weight))
FormulaTerm
Response:
  scale_factor(unknown)
Predictors:
  1
  avg_power_clean(unknown)
  hamming_distances(unknown)
  first_weight(unknown)
  second_weight(unknown)
  avg_power_clean(unknown) & hamming_distances(unknown)
  avg_power_clean(unknown) & first_weight(unknown)
  avg_power_clean(unknown) & second_weight(unknown)

julia> advanced_formula = @formula(scale_factor ~ 1 + avg_power_clean *
           (hamming_distances +
               first_weight + second_weight
               + first_mnemonic + second_mnemonic
           ))
FormulaTerm
Response:
  scale_factor(unknown)
Predictors:
  1
  avg_power_clean(unknown)
  hamming_distances(unknown)
  first_weight(unknown)
  second_weight(unknown)
  first_mnemonic(unknown)
  second_mnemonic(unknown)
  avg_power_clean(unknown) & hamming_distances(unknown)
  avg_power_clean(unknown) & first_weight(unknown)
  avg_power_clean(unknown) & second_weight(unknown)
  avg_power_clean(unknown) & first_mnemonic(unknown)
  avg_power_clean(unknown) & second_mnemonic(unknown)
```

Listing 3.3: Formulas for the inter instruction overhead linear model.

3.1.10. Trace selection

The last thing to consider is which instructions made up the program we need to estimate. This is a not trivial question, as it means to predict which instruction will be executed on the physical processor. In general, this is only known at run time: the emblematic example is an `if` clause whose condition is determined by user input. This is usually not the case for an embedded device, which is programmed to do one job in loop, usually with indirect interaction with the user. However, our analysis aims to be as general as possible, and even for embedded applications this problem may be undecidable until the real execution. We have therefore two possibilities: either we set up a test environment in which the application may run deterministically, or we simply don't try to predict the exact instruction sequence.

In the former case we can suppose to slightly modify the application, so that it no longer depends on the user or on random events, and execute it in a corresponding emulation environment to obtain a sequence of all the instructions executed. This is theoretically possible, but it has numerous drawbacks. First, we need an emulation environment corresponding to the processor under test, potentially supporting the guest operating system (OS) and the device peripherals. In this work we used the GNU project debugger (GDB) [24] and a modified version of QEMU [1] which works, but in general the support for ARM processors and bare metal application is extremely lacking. This is due to the fact that in general these environments are not required to be simulated: while this is generally true, it means that for lots of environments it would be needed to write a custom simulator to obtain the application trace. Since we would like to devise a method that requires the littlest intervention from the user, this is equivalent to say that some processors will not be testable until they are integrated in a free or open source tool like QEMU. Moreover, the profiled trace will not necessarily be the same of the actual application, so we may end up having a precise trace of a different application.

The remaining case is the one of not predicting the trace at all. What we can do in this case is to profile the source code, supposing each instruction will be executed exactly once. The underlying hypothesis is that a single basic block will either not be run or it will be run entirely. This means that when performing optimizations we can say that assigning a cost to the two versions of the basic block by looking at the instructions will probably reflect the actual power consumption. In this work we implemented both the techniques, but we would choose the latter as a default implementation in an hypothetical power-driven optimizing compiler.

In either case, to obtain the trace for the program under test we wrote two Julia functions

```
target remote 127.0.0.1:1234
set logging file /tmp/gdb.txt
set logging overwrite on
set logging enabled off
file example.elf
display/i $pc
break *main
continue
set $i=0
while ($i<500)
  ni
  set $i = $i + 1
end
set logging enabled on
bt
set $i=0
while ($i<4000)
  si
  set $i = $i + 1
end
quit
```

Listing 3.4: GDB command file when connecting to a QEMU simulated program.

to extract it. When working with the deterministic program in the simulation, the function `get_trace_gdb` first prepares a file to be read by GDB containing instructions about how to process the executing program. The Listing 3.4 shows an equivalent example of the result, assuming that the program is called `example.elf`. Line 1 connects GDB to a process exposed to localhost on port 1234: this is where QEMU will run. Then GDB will set up the logging of its session but it will keep it disabled for now (lines 2 to 4), and it will load the same object file running on the simulator (line 5). This may help to obtain more information if in future one would want to make GDB print it. Line 6 instructs GDB to display at each breakpoint and step the program counter, and format it as an instruction. Then it places a breakpoint upon entering function `main`, reaches it (line 8), and then uses an helper counter variable to skip the first 500 instructions (lines 9 to 13). The first instructions are related to the board boot process, do we are not interested in those, and their number can be customized using a Julia variable. Then GDB will turn on the logging (line 14), print a backtrace of the stack for debugging purposes (line 15), and then prints the following 4000 instructions (this number too can be customized). The version of GDB to call is obtained by the same bare-metal specific toolchain that provided the version of the GCC compiler. An example of what the log looks like is given in Listing 3.5: GDB prints out the program counter (corresponding to the address in

memory of the instruction), the location of the instruction (the function name and its offset from the start) and its textual representation. The QEMU process is no longer

```
#0  0x08000240 in main ()
0x080001fc in fac ()
1: x/i $pc
=> 0x80001fc <fac>:        push       {r7}
0x080001fe in fac ()
1: x/i $pc
=> 0x80001fe <fac+2>:        sub        sp, #20
0x08000200 in fac ()
1: x/i $pc
=> 0x8000200 <fac+4>:        add        r7, sp, #0
0x08000202 in fac ()
1: x/i $pc
=> 0x8000202 <fac+6>:        str        r0, [r7, #4]
```

Listing 3.5: Example extract of a GDB log.

needed and it is stopped, while the log is processed using regular expressions to extract the sequence of instructions. This sequence is finally decorated using the seen metadata functions, and the function returns a vector of DataFrame rows.

On the other hand, when extracting the sequence of instructions from the plain binary file, we act on the text representation obtained by the `objdump` tool (again, the specific toolchain version for simplicity). We first invoke `objdump` to dump all the functions in the code to a string: two fractions of function disassembly can be seen in Listing 3.6. We can see that `objdump` prints the address in memory of each function, its binary encoding in hexadecimal, the textual representation, and tries to place the labels indicating the starting points of the functions. Moreover, it highlights when it thinks a function has been called by decorating the branch instructions with the destination label. Using this information, we can parse the text to recursively build a graph of the called functions, starting from the `main` function or from a specified label. This allows us to insert in the trace exactly once all and only the functions that may be executed starting from the specified point. This may be useful if for some reason the compiler didn't optimize away the dead code, but knowing it will never be executed we would like to not add its score to the total. For debugging purposes the graph can be plotted, as shown in Figure 3.1.5. Here too a regular expression is used to extract address, encoding and instruction text from each line. Since each address acts as a key to identify each instruction (i.e., to each address corresponds at most one instruction), we can check that no instruction has been taken twice. Then, we can decorate the instructions with the metadata functions, and we can return the vector of rows, as we did in the simulated case.

```
08000232 <main>:
 8000232:       b580            push     {r7, lr}
 8000234:       af00            add     r7, sp, #0
 8000236:       f002 fac3       bl      80027c0 <HAL_Init>
 800023a:       f000 f805       bl      8000248 <SystemClock_Config>
 800023e:       2005            movs     r0, #5
 8000240:       f7ff ffdc       bl      80001fc <fac>

    ; [...]

080001fc <fac>:
 80001fc:       b480            push     {r7}
 80001fe:       b085            sub     sp, #20
 8000200:       af00            add     r7, sp, #0
 8000202:       6078            str     r0, [r7, #4]
 8000204:       2301            movs     r3, #1
```

Listing 3.6: Fractions of `objdump` output for the example program.

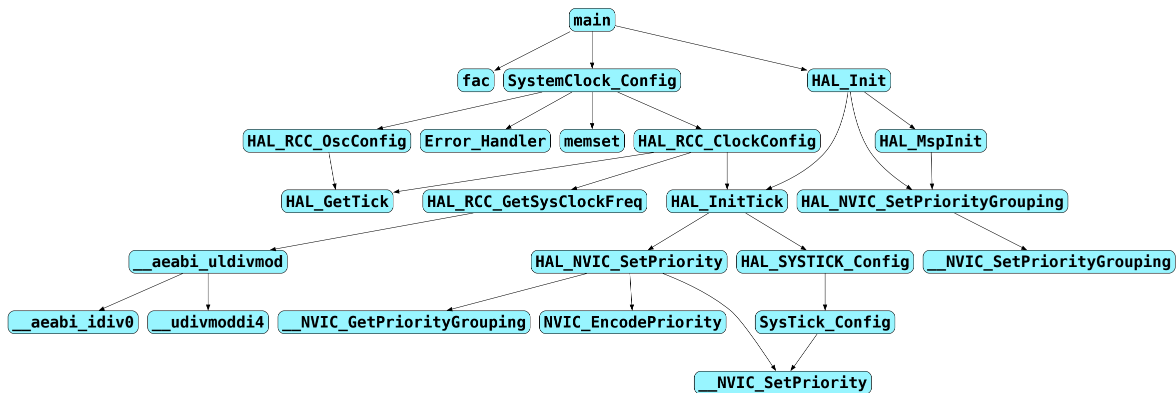


Figure 3.1.5: Call graph for the example program.

4 | Experimental results

We present the data obtained with the process described with two goals. First, we want to show the data exploration we performed in order to guide the analysis: this explains some of the design choices we made in constructing the models, and highlights some of the limitations of this experimental setup. Second, we show how good the oracle was in predicting power differences in some test programs.

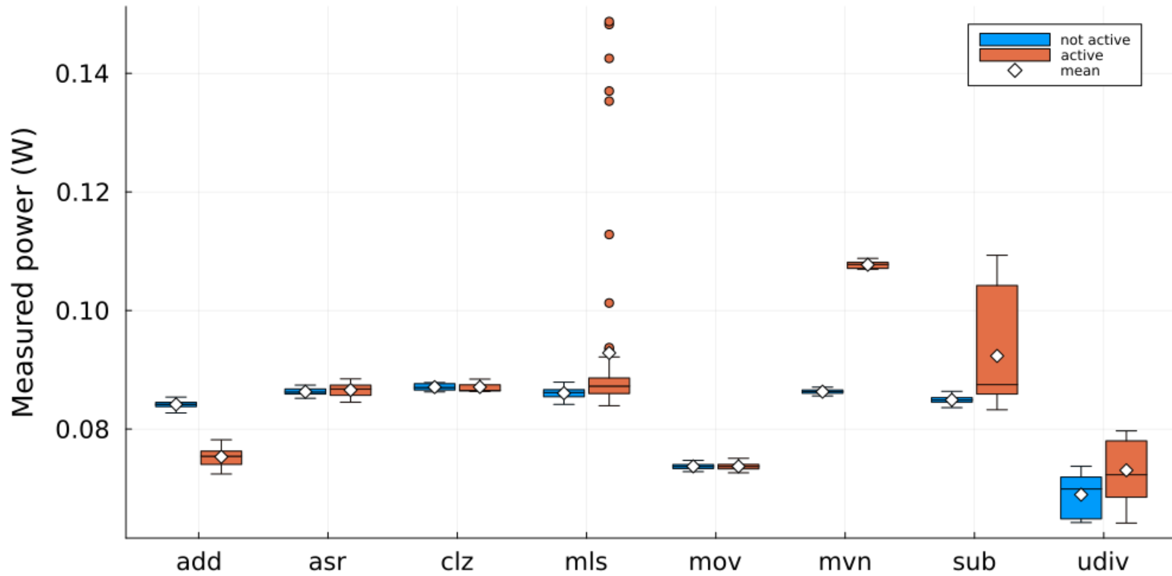
4.1. Data exploration

4.1.1. Assessing the regression variables effects

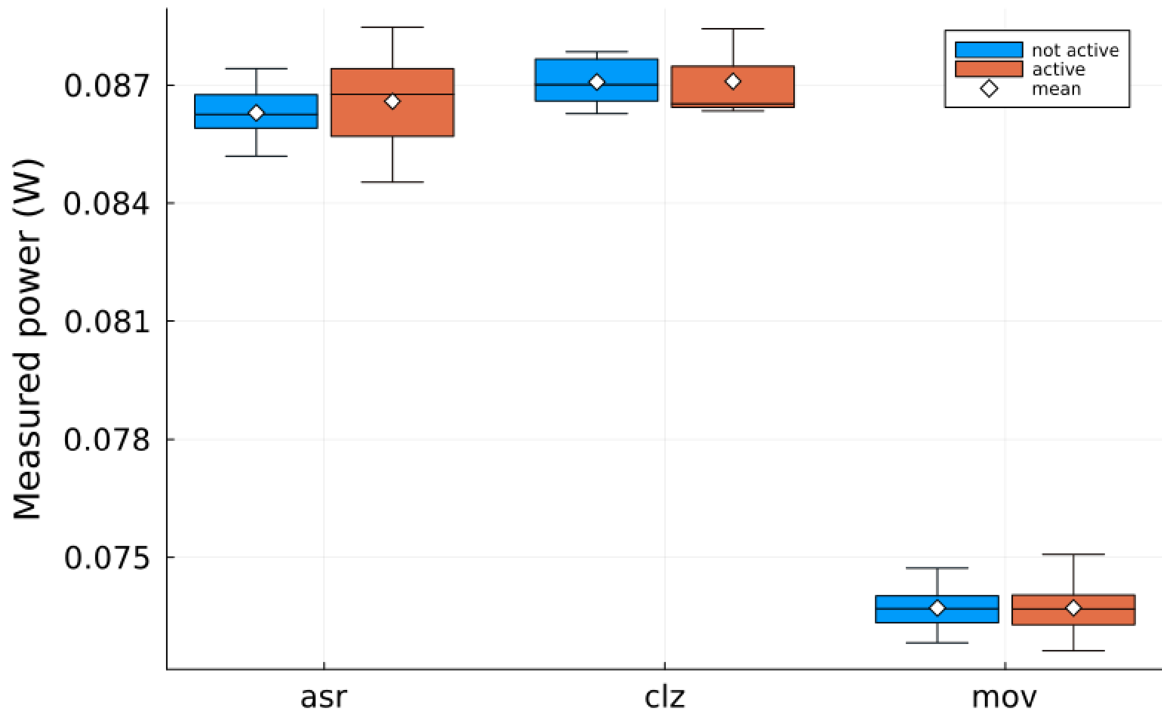
Our data exploration starts with merely observing the power plots produced by the Otii GUI. Albeit inaccurate, they started showing some of the behaviours that we later confirmed with a more accurate data analysis. In the following sections we show the effects that became the regression variable we used in the model. To analyse the effects as isolated as possible one from the other, when extracting the data to visualize we filtered it so to have all other variables fixed. For instance, when analysing the effect of the forwarding paths, we used only the instructions that did not use neither the conditional execution, nor the "`s`" flag, nor an immediate operand, nor the operand shift. The filters we applied are summarized in Table 4.1. The choice of using a positive or negative filter (i.e., whether to consider the instruction that presented the effect or that did not) was taken as to have the most significative dataset resulting. Having chosen in a couple of cases an inverse filter is not a problem, since our need here is to isolate the best we can the effects of single variables, and thus it does not invalidate the effect analysis.

Usage of the forwarding paths

As we anticipated in Section 3.1.8, we observed that the instructions that used the processor forwarding paths used different amount of energy with respect to those who did not. Figure 4.1.1a shows this effect on a selection of instructions.



(a) Selected instructions



(b) Particular

Figure 4.1.1: Effect of the CPU forwarding paths.

Table 4.1: Filters applied to the data points to isolate the tested effect. The symbol • means that we considered the instructions that did not have the fixed effect, while the symbol ○ means that we considered those who did.

Tested effect	Fixed effect				
	Forwarding	APSR	Conditional	Shift	Immediate
Forwarding		•	•	•	•
APSR	•		•	•	○
Conditional	•	•		•	•
Shift	○	•	•		•
Immediate	•	•	•	•	

We observe different effects on the power consumption:

- mean power consumption increased for all instructions except for `add`
- instructions `asr`, `mls`, `mvn`, `sub` and udiv all increased their median consumption, while `clz`, `add` and `mov` decreased it
- instructions `add`, `asr`, `mls` and `sub` increase their interquartile range (IQR), while others do not (at least, not significantly)
- instruction `mls` introduces a fair number of outliers that shift the mean

These effects may be due to the different areas of the processor that were activated, and to the fact that always overwriting the content of one of the source registers we are actually constantly changing the state of the signals in the processor, thus introducing a much higher switching activity. A summary of the effects of the variable is shown in Table 4.2.

Moreover, in Figure 4.1.1b we show a close-up of instructions `asr`, `clz` and `mov`, which do exhibit differences, but of much littler magnitude compared to other ones like `mvn`.

Writing of the program status register

One of the features of the ARM ISA is that normal instructions do not update the content of the application program status register (APSR). To do so, it is required to add the suffix "`s`" to the instruction: for instance, instruction `add` computes the sum of two binary numbers; `adds` does the same but updating the APSR flags. Those flags are then used to read the result of comparisons, as they signal whether two numbers were equal, if an overflow occurred etc. From an hardware point of view, we can assume that setting the

Table 4.2: Summary of statistics changes for the usage of forwarding paths. The symbol ● means that the statistic for that instructions increased when the effect is active, ○ means it decreased, and ◐ means it stayed within a 5 % range.

Statistic	Instruction							
	`add`	`asr`	`clz`	`mls`	`mov`	`mvn`	`sub`	`udiv`
Mean	○	◐	◐	●	◐	●	●	●
Median	○	◐	◐	◐	◐	●	◐	◐
Variance	●	●	●	●	●	●	●	●
IQR	●	●	◐	●	●	●	●	●

register will activate other paths in the processor, appointed to forward the flags produced by the ALU to the APSR. Thus different areas of the processor will consume power, and we expect to see this difference in our data. Following the idea of previous section, we filtered out unrelated variables. The effect is shown in Figure 4.1.2 (again, on a selection of instructions).

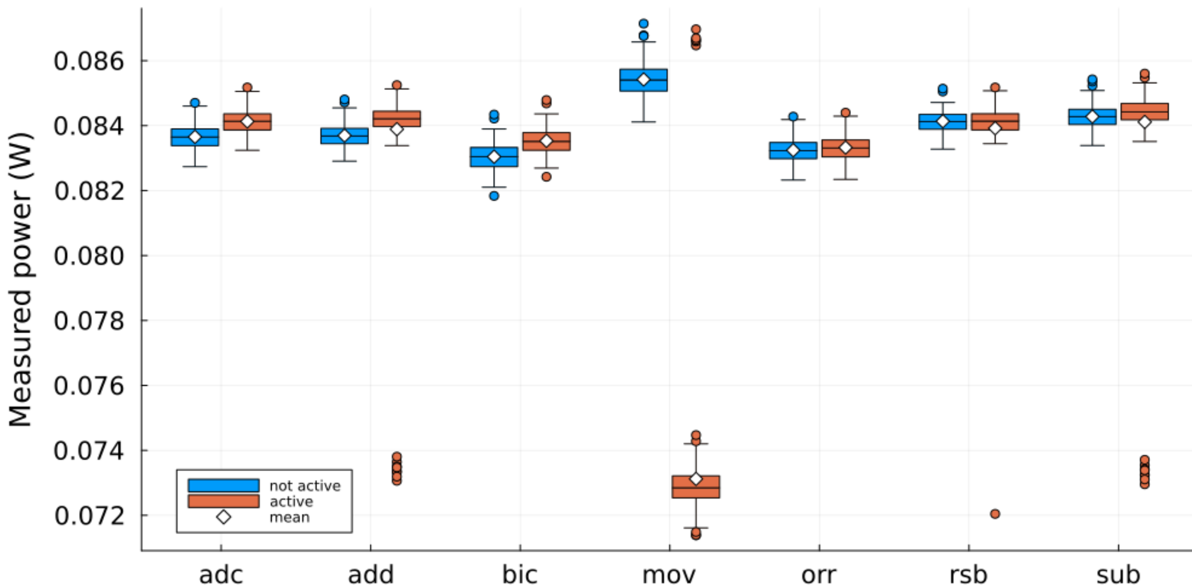


Figure 4.1.2: Effect of updating the APSR ("`s`" flag).

Here too we can see different effects on different instructions:

- mean power consumption increased for instruction `adc`, `add`, `bic` and `orr`, and it decreased for all the others

- the IQR stays constant, but some instructions (`add`, `mov`, `rsb` and `sub`) introduce outliers
- median power consumption increased for all instructions, except `mov`

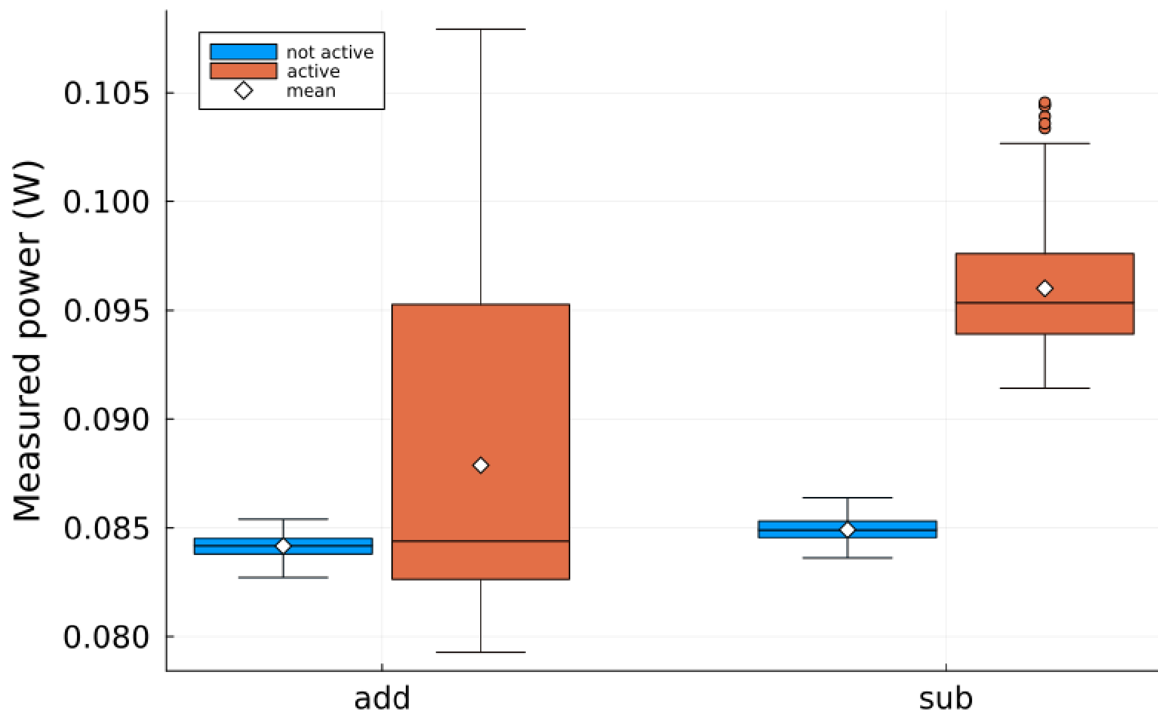
Instruction `mov` is the only one from the ones we tested that exhibits such a great difference. A summary of the effects introduced can be seen in Table 4.3.

Table 4.3: Summary of statistics changes for the updating of the APSR ("`s`" flag). The symbol ● means that the statistic for that instructions increased when the effect is active, ○ means it decreased, and ◐ means it stayed within a 5 % range.

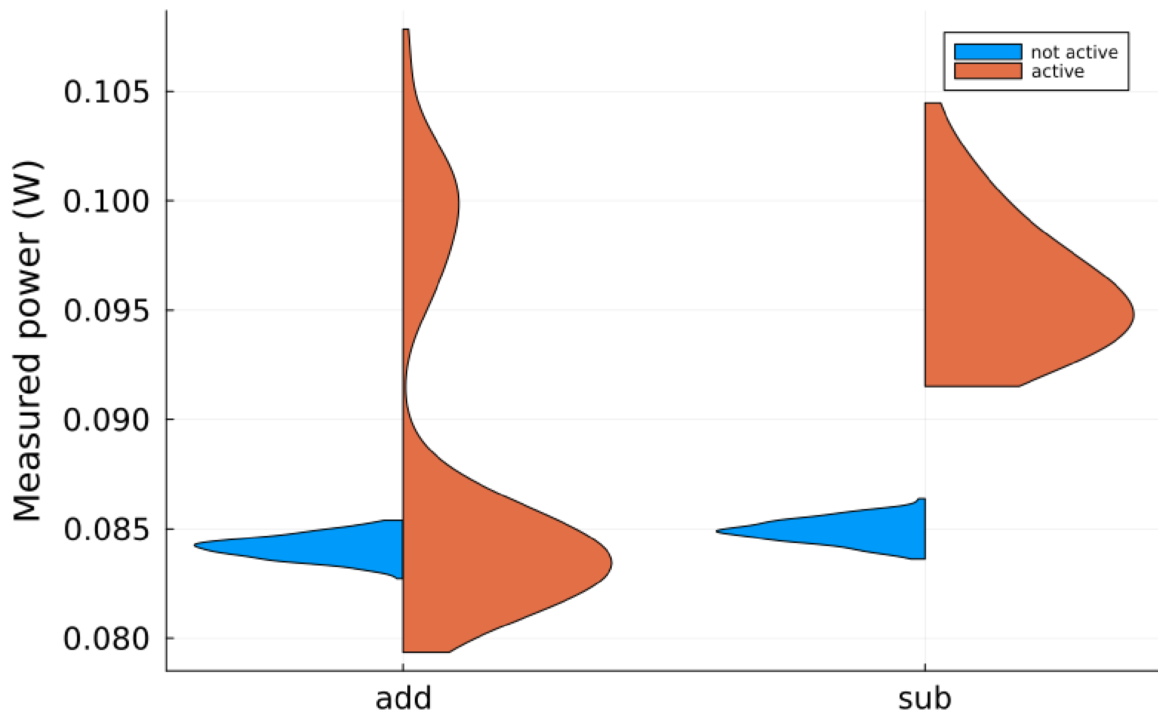
Statistic	Instruction						
	`adc`	`add`	`bic`	`mov`	`orr`	`rsb`	`sub`
Mean	◐	◐	◐	○	◐	◐	◐
Median	◐	◐	◐	○	◐	◐	◐
Variance	○	●	○	●	●	●	●
IQR	◐	◐	○	◐	◐	●	●

Conditional execution

The normal ARM instructions are executed one after the other. To change the control flow, one can use branches or conditionally executed instructions. The latter means to add to the instruction mnemonic code a suffix representing a condition, like the instruction `addeq` that executes only if the APSR flag Z is set to 1 (indicating the two numbers of the last comparison were equals). In general, when the condition is true, the instruction is executed, otherwise it is skipped (or, equivalently, it is read as a `nop`). Since we tested this effect only on instructions `add` and `sub`, we show the plot only for those, in Figure 4.1.3. To collect the data for this particular setting, we first executed a `cmp` or equivalent instruction to set the APSR in a state that would permit the execution of the instruction, then we injected such instruction as usual. To better show the difference in the distributions, we plotted both the box plot and the violin plot for the power distribution. We can observe that the mean power consumption increased greatly, just like the IQR; the median on the other hand increased significantly just for the `sub` instruction. Again, the effect is not constant between the two instructions. A summary is shown in Table 4.4.



(a) Box plot of the effect



(b) Violin plot of the effect

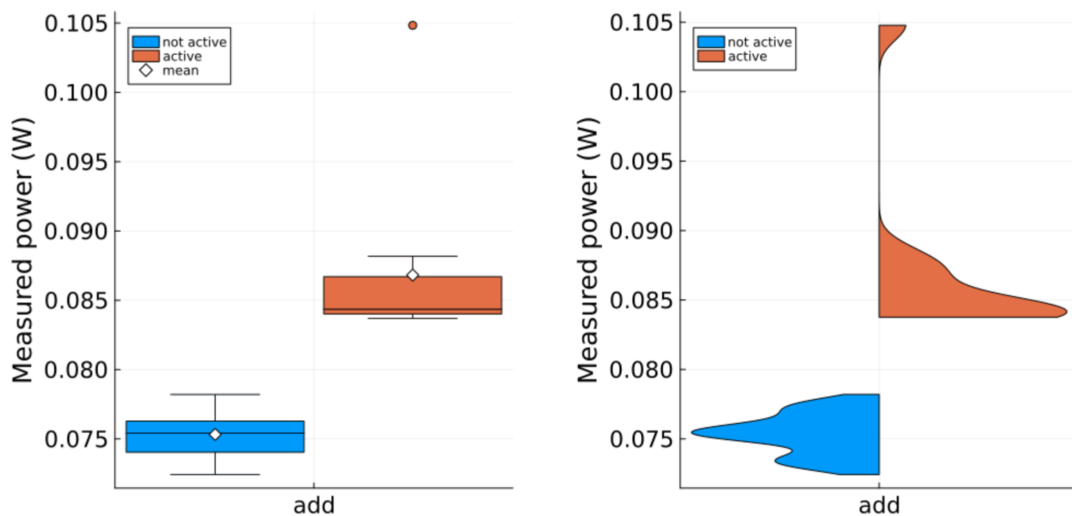
Figure 4.1.3: Effect of using the conditional execution.

Table 4.4: Summary of statistics changes for the usage of conditional execution. The symbol ● means that the statistic for that instructions increased when the effect is active, ○ means it decreased, and ◐ means it stayed within a 5 % range.

Statistic	Instruction	
	`add`	`sub`
Mean	◐	●
Median	◐	●
Variance	●	●
IQR	●	●

Barrel shift

We also investigated the ability of the ARM processors to shift their second operand (a feature sometimes called "barrel shift"). This time, we only tested as a sample the `add` instruction. The result of the usual analysis is shown in Figure 4.1.4. This time all the



(a) Box plot of the effect (b) Violin plot of the effect

Figure 4.1.4: Effect of using the barrel shift.

statistics but the IQR increased. However, the variable changes the statistical distribution, and introduced an outlier that shifts the mean.

Usage of an immediate operand

A last thing we may want to test is whether the power consumption changes when the second operand is an immediate with respect to when it is not. This is however just a circumstantial analysis: when the operand is a register, power consumption may depend greatly on the content of the register, thus the difference may change depending on the status of the processor registers. However, to see if it is worth including this difference as a regression variable, we can observe the data, as shown in Figure 4.1.5a. We see that:

- all instructions but `mov` decrease their mean and median consumption
- the IQR tends to slightly decrease, but for some instructions (particularly `cmp`, `mov` and `ror`) the immediate operand introduces outliers

Instructions `cmp` and `mov` are worth a particular mention, as for the former are introduced far outliers, while the latter presents a huge gap in mean power consumption, and it is the only one from the selected to show an increase in such sense (an effect similar to what the same `mov` instruction did when observing the effect of updating the APSR). Also, instruction `vmov` decreased greatly its IQR.

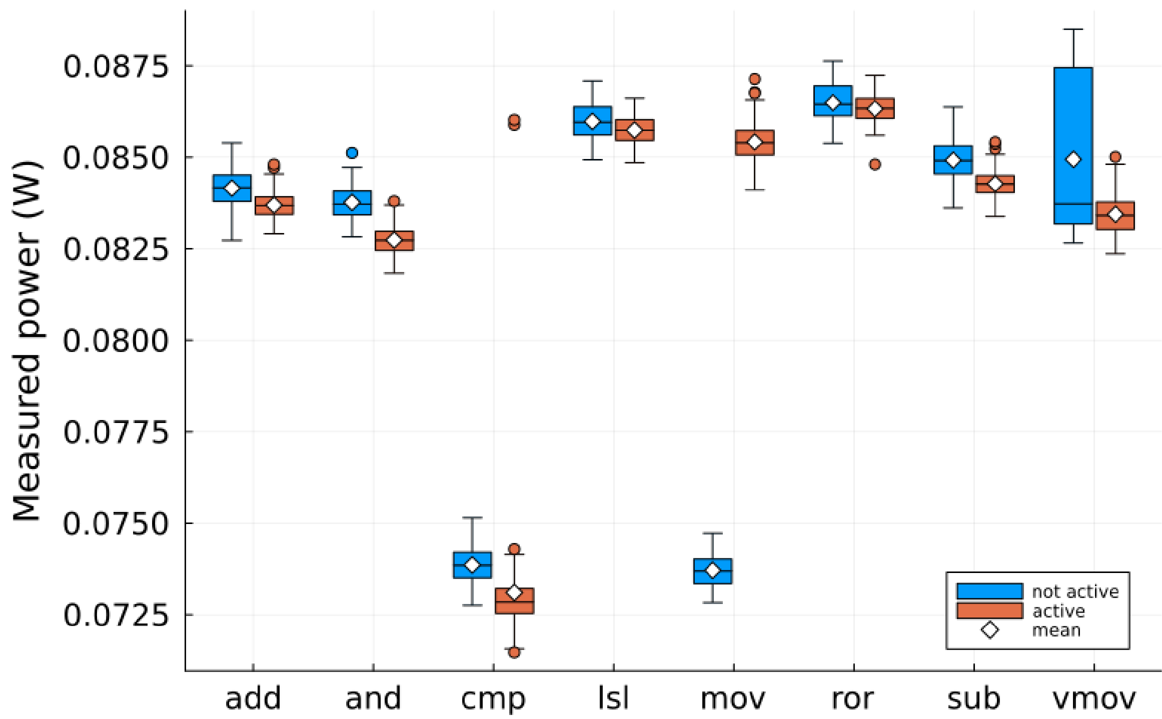
Figure 4.1.5b shows the other instructions magnified to better appreciate their effect, and a summary of the statistics changes is shown in Table 4.5.

Table 4.5: Summary of statistics changes for using an immediate operand. The symbol ● means that the statistic for that instructions increased when the effect is active, ○ means it decreased, and ◐ means it stayed within a 5 % range.

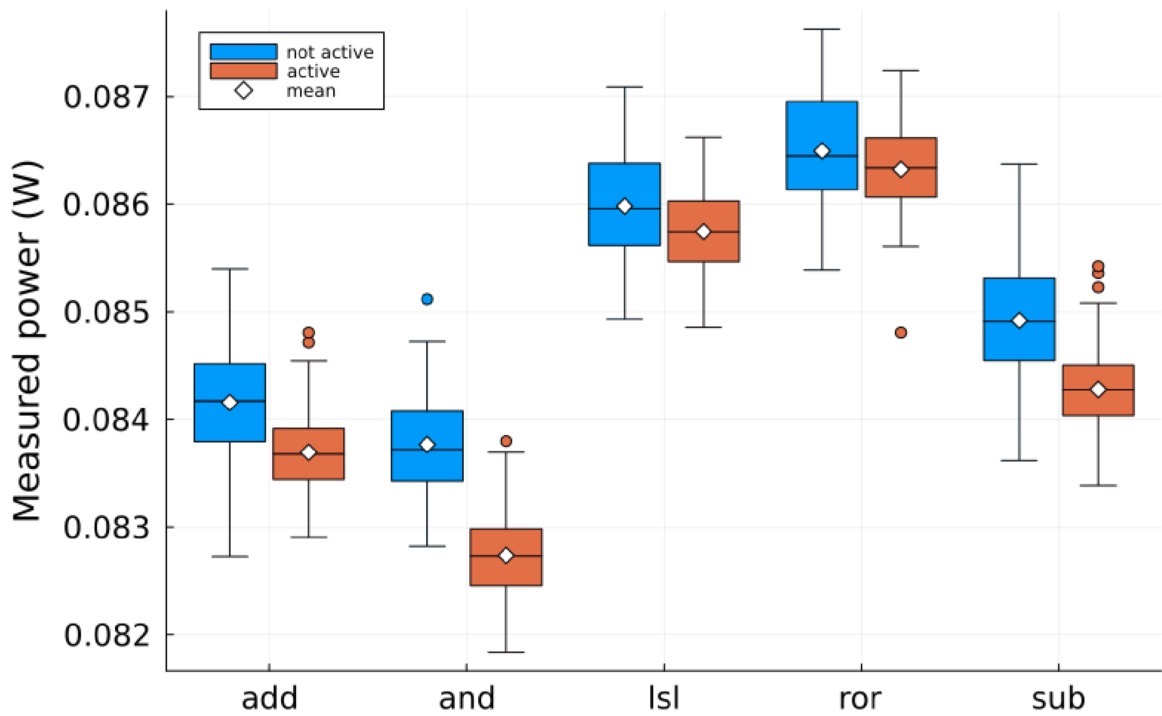
Statistic	Instruction							
	`add`	`and`	`cmp`	`lsl`	`mov`	`ror`	`sub`	`vmov`
Mean	◐	◐	◐	◐	●	◐	◐	◐
Median	◐	◐	◐	◐	●	◐	◐	◐
Variance	○	○	●	○	●	○	○	○
IQR	○	○	◐	○	◐	○	○	○

Conclusions

As we have seen in the past examples the effect relative to the regression variables we chose does have an effect on the power consumption. However, such effect is not constant across all the instructions. This means that it is better to correlate the usage of the paths with the instruction mnemonic to obtain an higher precision. When the first model fails



(a) Selection of instructions



(b) Particular of most similar instructions

Figure 4.1.5: Effect of using an immediate value as second operand.

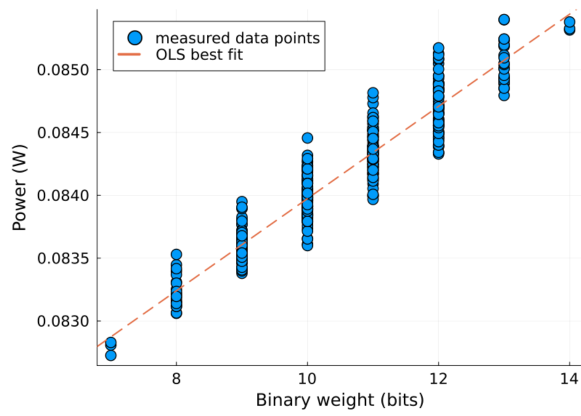
and we resort to the second, simpler model, or we use a constant effect (as we explained in Section 3.1.8), the precision of the result may significantly be reduced.

4.1.2. The binary weight dependency

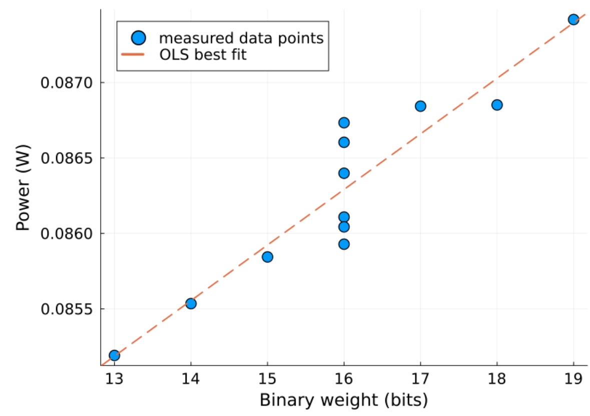
As we mentioned in Section 2.2.3, we expect the power consumption to depend significantly on the binary weight of the signal flowing in the processor. Since we cannot know, in general, the internal state of the CPU at runtime (it is possible only in a completely deterministic case, that is when no data is collected from external agents such as the user or a random source), we focus our analysis on the binary weight of the instruction currently being executed. We do this with an analysis similar to the previous: we filter out all the irrelevant instructions, i.e. the ones presenting the effects we do not want to consider, then we plot the data. In the following we present a few instructions where all the additional effects have been filtered out, except for the usage of immediate values for the second operand: for that, we present examples from both the datasets.

We start with Figure 4.1.6, where we plotted instructions that used all register operands. Some instructions data align nicely onto a line (like `add`), others exhibit a linear dependency but more sparse (`asr`). The size of the data set is important: with fewer data points to perform the regression on, we are less confident about the goodness of the prediction, and we can even question if a linear dependency is present at all. For instance, instruction `umull` may have a linear dependency but also a relevant heteroscedasticity. This may mean that there is at least one other variable that we have not captured [4], or that we simply ended up with a too little dataset, and this apparent behaviour showed up by chance. Anyway, even in case the data do show heteroscedasticity, the linear regression model will still give unbiased predictions of the model parameters, so we can ignore the phenomenon. These errors will not statistically affect real programs, as they are large enough that the errors effect will be diluted. If the dataset is degenerately small, such as for instruction `bx`, we can have a trivially very close linear approximation (here we have just 3 data points), but we expect it to be very fragile, in the sense that new values may differ significantly from the predicted ones. Other instructions, such as `smlal` and `vpop`, present very sparse dataset, so the linear dependency is little to non-existent: however, the linear regression tool is robust to this event, in the sense it naturally tends to assign a value of 0 to the coefficients of non correlated variables, thus not invalidating the model.

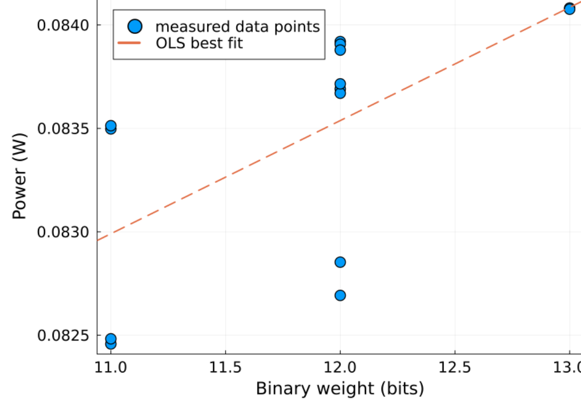
We can observe other examples when using immediate values, as shown in Figure 4.1.7. Here too some instructions (like `add`) present a good correlation, others are sparse (`ldrb`), and others have very few data points (`blx`). There are also instructions that form clusters



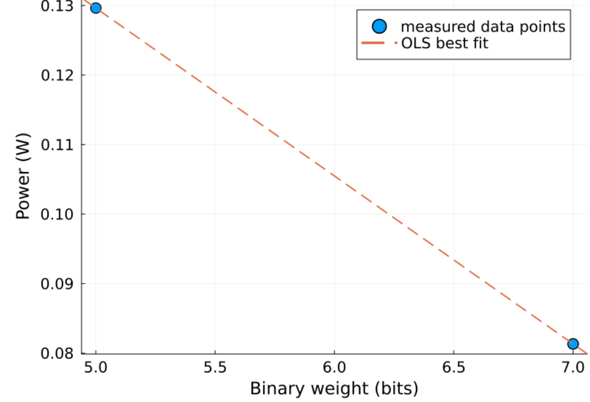
(a) `add` instruction



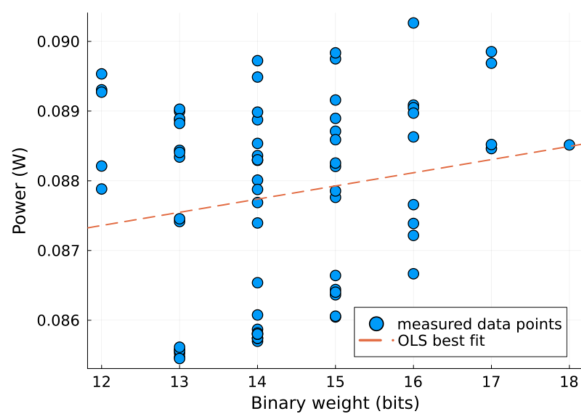
(b) `asr` instruction



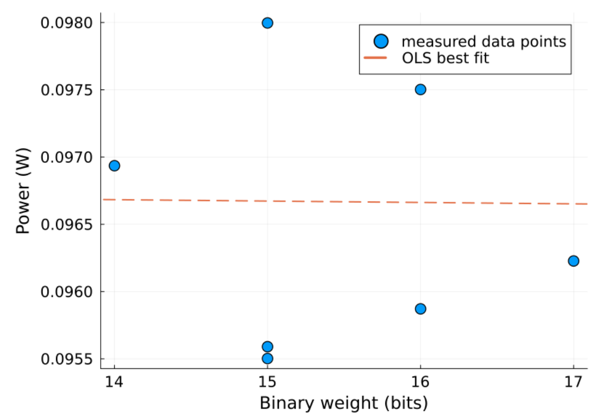
(c) `umull` instruction



(d) `bx` instruction



(e) `smlal` instruction



(f) `vpop` instruction

Figure 4.1.6: Dependency of power consumption by the binary weight, for a selection of instructions using all register operands.

of points, like `bl`, `cmp` and `ldr`. This suggests that there are more hidden variables that can be identified within the data, and that may be discovered with the help of machine learning techniques, as stated before. Such analysis however is over the scope of our work.

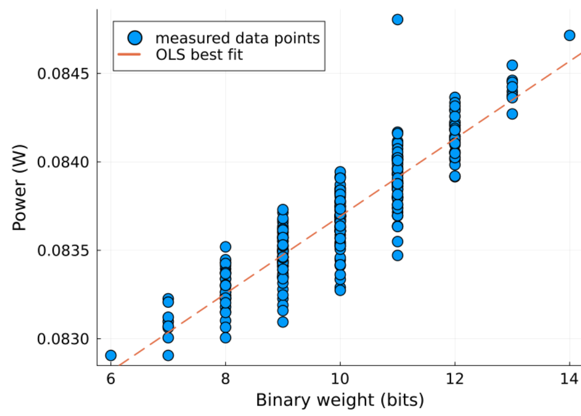
An attempt we can do to soften our model anomalies is about the instruction working with the machine memory. We can expect the bad performance of load and store instructions: their effects are also determined by the content of the memory and more importantly by the current state of the processor cache, which we cannot know before runtime. A possible patch to the problem may be correlating their power consumption with the distance in bytes from the address of the current instruction. In this way, when we read/store on a memory location "near" from where the execution is, we can suppose the probability of a cache miss is lower, since we know for a fact that close areas of memory are already in the cache (since we are reading the code from there); dually, we can assign an higher cost to higher distances. In the code of the model we kept track of this when calculating the amount of the immediate operand of a given instruction (see Listing B.1 at page 90): when we are decorating a branch instruction, the value we store is actually the difference between the destination and the current address. We can see this dependency on Figure 4.1.8: even if a correlation seems to be present, it does not appear to be close to linear. This may be due to the fact that, after the cache miss threshold has been exceeded, there is no dependency on how far the new cache block will be loaded. An activation function may be a better fit than a linear dependency, but this would require to further modify our model. Therefore, we leave this as future work.

4.2. Results

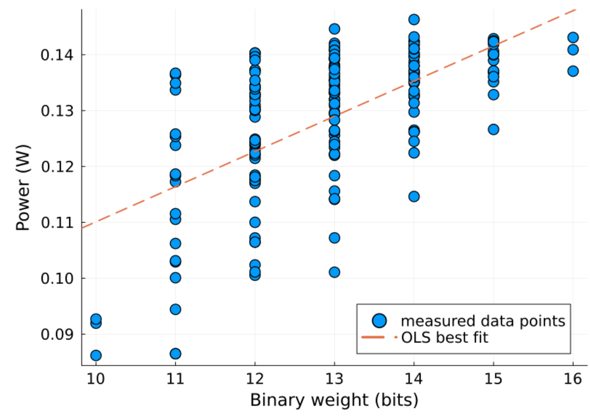
We now present the results of our oracle, both with respect to the train data (Section 4.2.1) and when doing validation on real programs (Section 4.2.2). The latter has been conducted in two scenarios, as we introduced in Section 3.1.10: by simulating a completely-deterministic program (without user input), or by estimating a binary code.

4.2.1. Goodness of the linear models

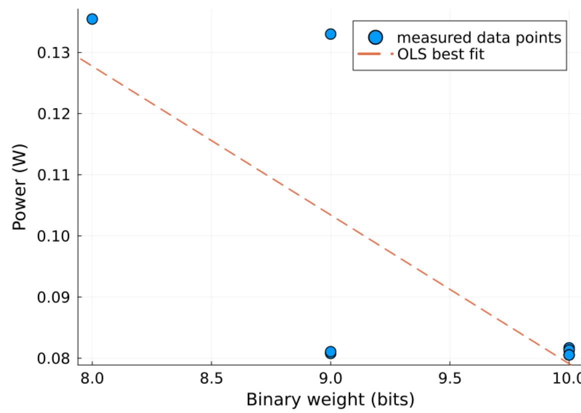
The complete linear model describes sufficiently well the input train data. A common statistic describing how much of the data behaviour is captured by the model is the R^2 index [4]: for this model it's 0.7938, meaning that 79.38 % of the input behaviour is explainable with the model. We can then compute the mean absolute percentage



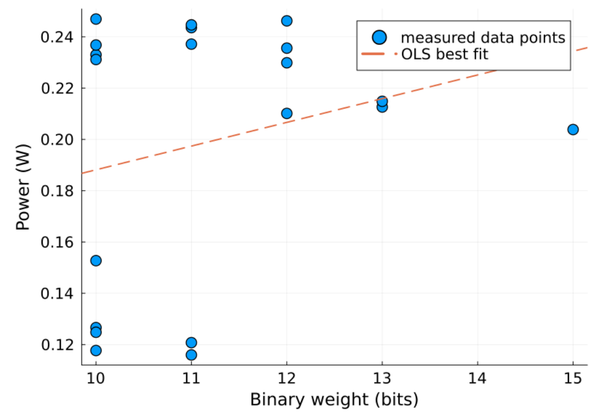
(a) add instruction



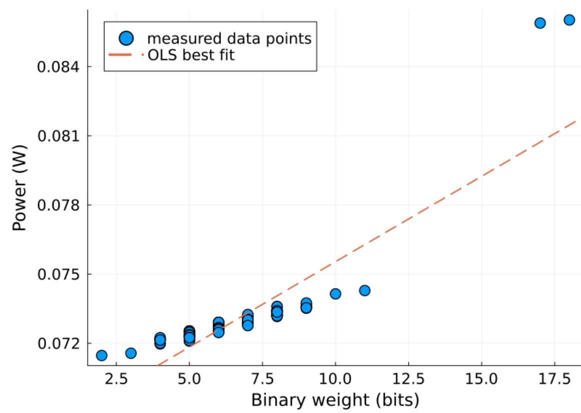
(b) ldrb instruction



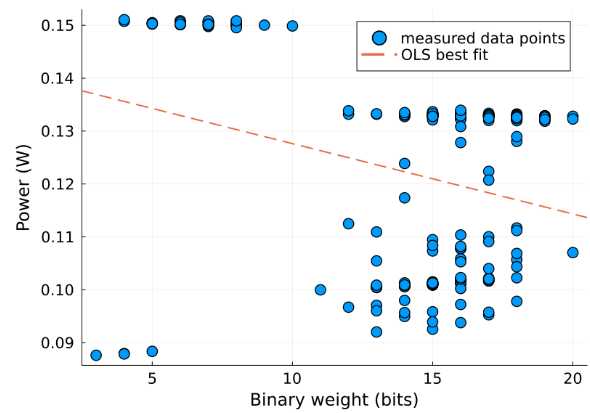
(c) blx instruction



(d) bl instruction

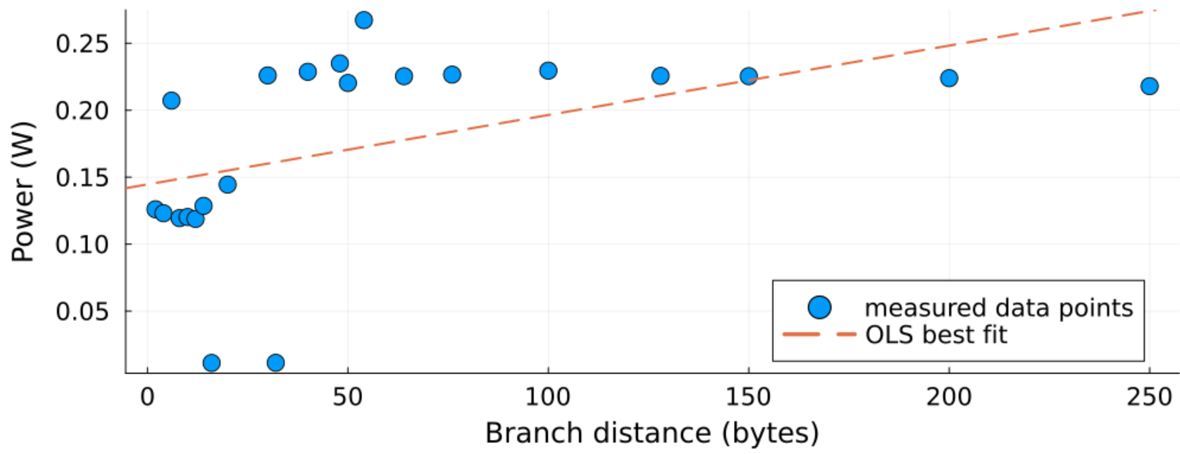


(e) cmp instruction

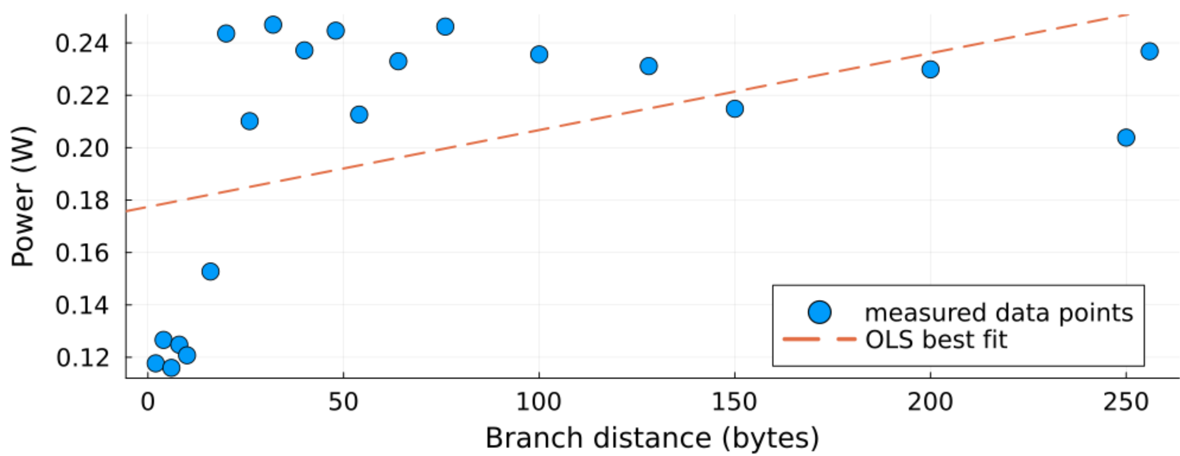


(f) ldr instruction

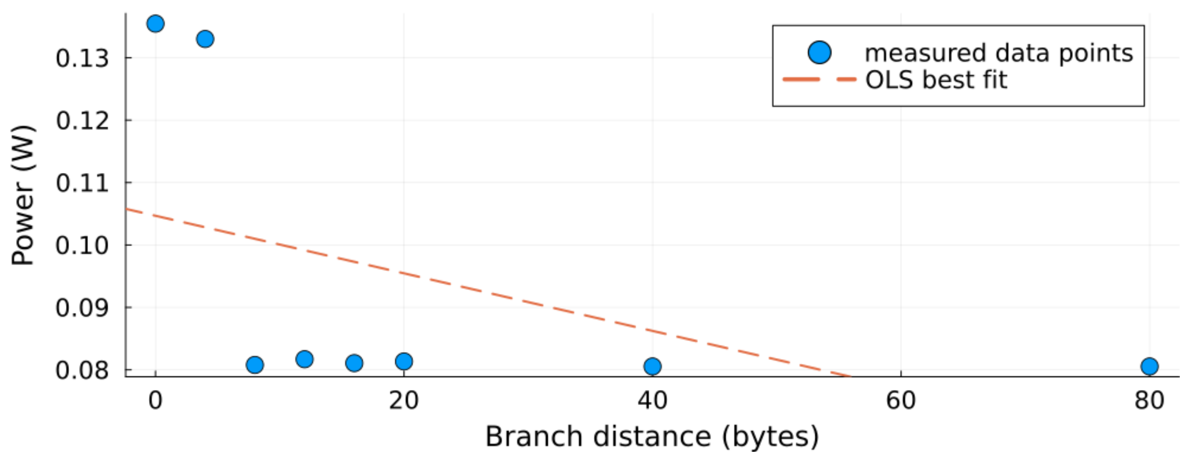
Figure 4.1.7: Dependency of power consumption by the binary weight, for a selection of instructions using all immediate operands.



(a) `b` instruction



(b) `bl` instruction



(c) `blx` instruction

Figure 4.1.8: Dependency of the power consumption on the branch distance.

error (MAPE) with the usual formula

$$\text{MAPE} = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{y_t - \hat{y}_t}{y_t}\right| \tag{4.2.1}$$

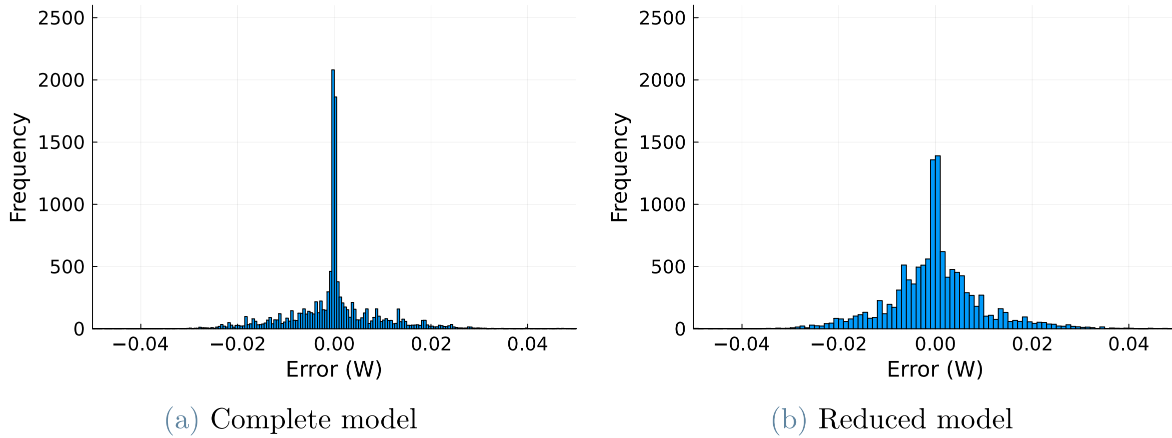
where the y_t are the measured powers, the $\hat{y}_t$ are the predicted powers, and n is the number of observations. For the complete model, we obtain a value of around 6.4 %. Then we can perform a 10-fold cross validation, calculating the MAPE for each subset. The obtained values may fluctuate due to the random shuffling we do before partitioning the train dataset; however their value have a mean of 6.6 %. Lastly, we can perform a t-test to check whether the regression coefficients are statistically different from 0 (this was done automatically by the statistical package we used). More precisely, the null hypothesis is that the coefficient is 0, and the alternative hypothesis is that it is different from 0. For the complete model, we observe that the 26.92 % of coefficients has a p-value above the usual significance level of 0.05.

On the other hand, the reduced model generally has worse indexes, being less complete. The R^2 index is 0.7711 (77.11 %), the total MAPE is 7.27 %, the 10-fold mean MAPE is 7.41 %, and the fraction of coefficients whose p-value is over 0.05 is 53.88 %.

We can also observe some behaviour of the models through plots. In Figure 4.2.1 we see the error distributions for the two models. As we see, the complete model is more accurate, while the errors of the reduced one are more spread out. On the other hand, in Figure 4.2.2 we see the model instructions in a plane whose dimensions are the measured and predicted power values. The turquoise line represents the equation $y = x$, that is when the predicted value coincide with the measured one (and thus the error is null). It is difficult to appreciate the difference on the plot, but the numeric error values confirm that the reduced model is a little more spread out.

This kind of error exploration can be done also on the single instructions. We can see some examples in Figure 4.2.3: some instructions are almost equally predicted by the complete and the reduced model (instruction `add`, Figures 4.2.3a and 4.2.3b), others are accurately predicted by the complete model, but not by the reduced one (instruction `and`, Figures 4.2.3c and 4.2.3d), and some are not accurately predicted by neither of the two (instruction `vdiv`, Figures 4.2.3e and 4.2.3f).

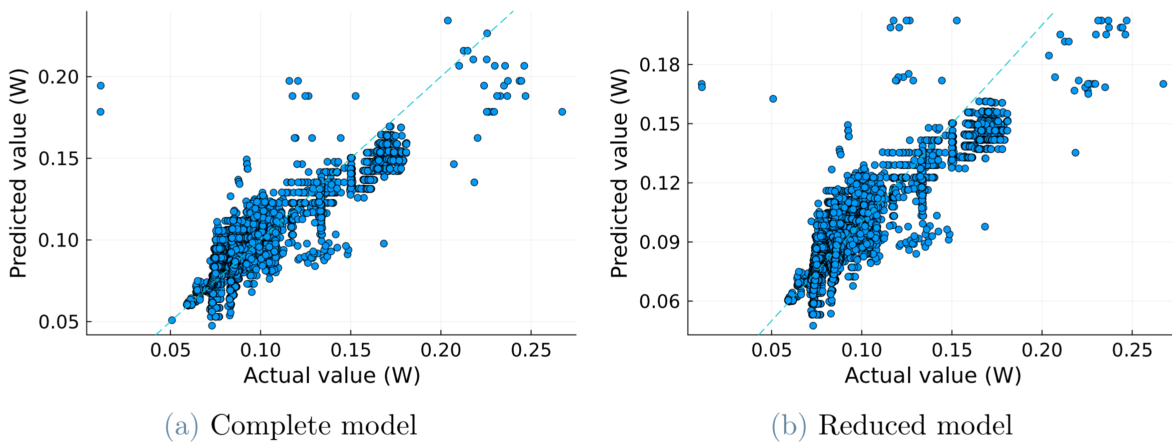
4.2.2. Model validation



(a) Complete model

(b) Reduced model

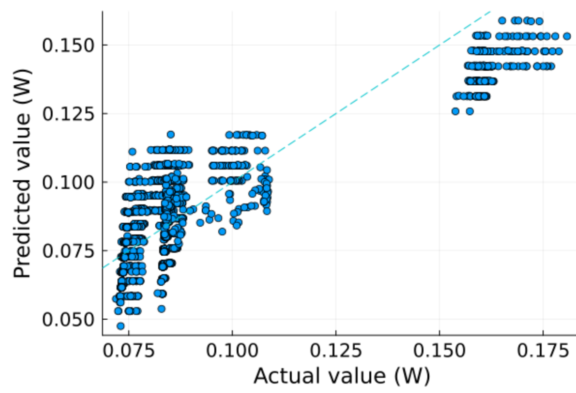
Figure 4.2.1: Histogram of error distributions of the linear models.



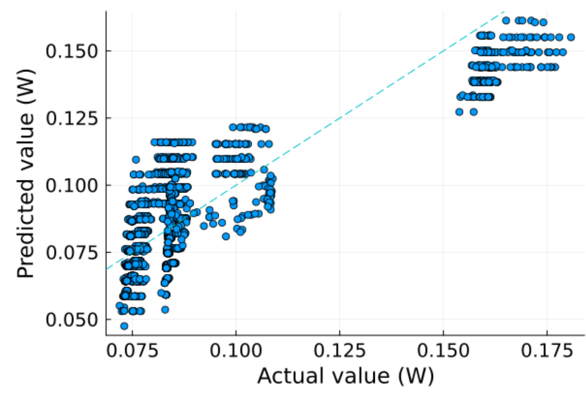
(a) Complete model

(b) Reduced model

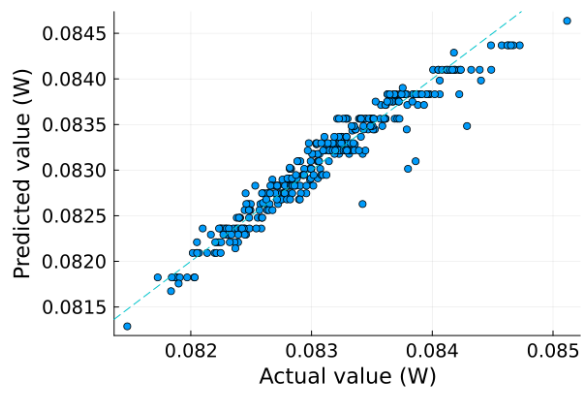
Figure 4.2.2: Difference between measured and predicted power values for the linear models. The turquoise line indicates where the two coincide, i.e. when the prediction is equal to the measured value.



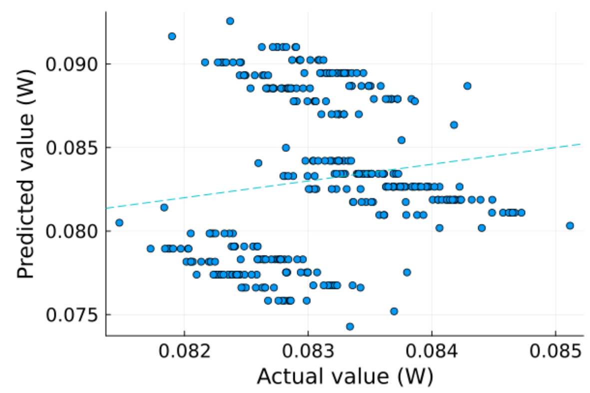
(a) `add`, complete model



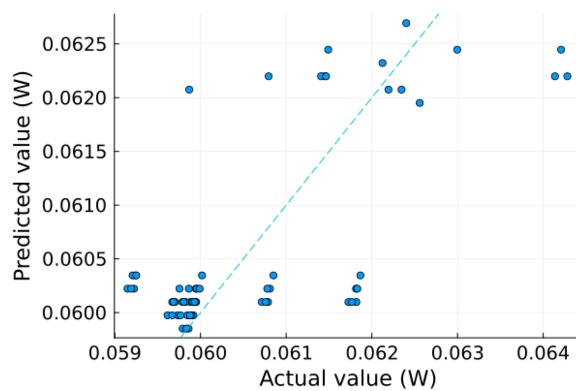
(b) `add`, reduced model



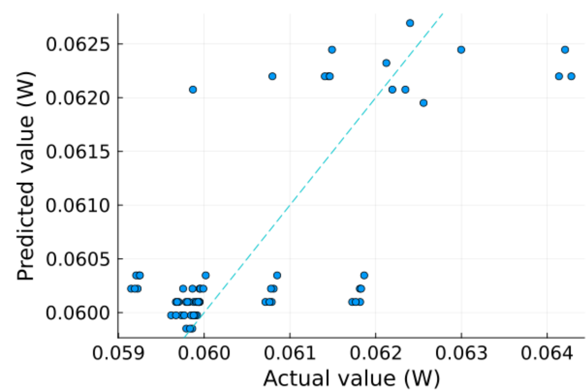
(c) `and`, complete model



(d) `and`, reduced model



(e) `vdiv`, complete model



(f) `vdiv`, reduced model

Figure 4.2.3: Difference between measured and predicted power values for a few instructions. The turquoise line indicates where the two coincide, i.e. when the prediction is equal to the measured value. On the left is shown the complete model, on the right the reduced one.

Simulating a deterministic program

For this use case, we needed to run a program that requires no user interaction or external inputs. We wrote simple examples of such a program, as we:

- calculated the nth Fibonacci's number recursively
- calculated the nth Fibonacci's number iteratively
- calculated the factorial of a number n recursively
- calculated the factorial of a number n iteratively
- calculated the sum of the first n natural numbers iteratively

All the source codes for these simple programs are shown in Listing 4.1. The functions have been tested by being called in the main infinite loop provided by the firmware C code; then, their power consumption has been measured like we previously did when measuring the instructions. A copy of each program binary has been saved to extract the trace with QEMU, as we explained in Section 3.1.10. The only other modification to the firmware we made was to disable the floating point unit, as the QEMU emulated machine had a bug that prevented using it and in our programs was not needed anyway.

The heat map in Figure 4.2.4 shows the difference of the powers consumptions measured when the programs were running on the board. The x and y axis (respectively growing to the right and to the bottom, as matrix indexes) correspond to the programs tested, i.e. the ones shown in Listing 4.1, each one called with the argument ranging in the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30\}$. Since reporting each program name would have turned out too cluttered, we marked just some of them. The meaning is that the colour of the square in position (x, y) reflects the difference $\Delta P = P_y - P_x$ (i.e. $row - column$), as indicated on the colour bar, measured in W. Similarly, Figure 4.2.5 does the same but for the power estimate we obtained from our oracle. In both cases the order of the programs on the axes is the same, that is by function first and argument then. As natural, the maps are antisymmetric with respect to the main diagonal. Moreover, we can see relatively distinct clusters of values, partitioning the maps in a 5×5 grid, and some more noticeable values at the borders of such partitions. This means that the power consumption is primarily determined by the function implemented (i.e., by the code running), and that the data signals flowing in the processor only partially contribute. The input values however are not irrelevant: the border values are the ones that trigger different control flows in the application. For instance, when evaluating the `fact` (recursive) function (Listing 4.1a) with `0` as argument, the condition at line 2 is true from the start, therefore line 3 is executed, and the `else` block is never entered, and the recursion does not actually take place. On

```
1 int fact(int n) {
2   if (n <= 1) {
3     return 1;
4   } else {
5     return n * fact(n - 1);
6   }
7 }
```

(a) Factorial (recursive).

```
1 int fib(int n) {
2   if (n <= 1) {
3     return 1;
4   } else {
5     return fib(n - 1) + fib(n - 2);
6   }
7 }
```

(b) Fibonacci (recursive).

```
1 int fact(int n) {
2   int res = 1;
3   for (int i = n; i > 0; i--){
4     res *= i;
5   }
6   return res;
7 }
```

(c) Factorial (iterative).

```
1 int sum(int n){
2   int res = 0;
3   for(int i = 1; i <= n; i++){
4     res += i;
5   }
6   return res;
7 }
```

(d) Sum of natural numbers (iterative).

```
1  int fib(int n) {
2    int fib_2 = 1;
3    int fib_1 = 1;
4    int fib_0 = 2;
5
6    if (n <= 1) {
7      return 1;
8    } else {
9      for (int i = 2; i < n; i++){
10       fib_2 = fib_1;
11       fib_1 = fib_0;
12       fib_0 = fib_2 + fib_1;
13     }
14     return fib_0;
15   }
16 }
```

(e) Fibonacci (iterative).

Listing 4.1: Deterministic programs.

the other hand, when calling the function with an argument of `10`, line 5 is executed many times. The net effect is that the difference between programs `fac_rec_#1` and `fac_rec_#2` is 4.084×10^{-3} W, while the difference between `fac_rec_#9` and `fac_rec_#10` is just -4.9×10^{-5} W. To show the complete data of an example program, we reported in Table 4.6 all the results of recursive factorial binaries.

Table 4.6: Results for the recursive fact function, measured from the trace provided by QEMU and GDB.

Program	Measured (W)	Predicted (W)	Error (%)
`fac_rec_#0`	0.121 555	0.074 372	−38.82
`fac_rec_#1`	0.121 587	0.074 372	−38.83
`fac_rec_#2`	0.117 503	0.099 467	−15.35
`fac_rec_#3`	0.116 771	0.107 857	−7.63
`fac_rec_#4`	0.116 257	0.111 863	−3.78
`fac_rec_#5`	0.116 123	0.114 325	−1.55
`fac_rec_#6`	0.115 823	0.116 345	0.45
`fac_rec_#7`	0.115 683	0.117 124	1.25
`fac_rec_#8`	0.115 455	0.118 032	2.23
`fac_rec_#9`	0.115 474	0.118 74	2.83
`fac_rec_#10`	0.115 523	0.119 347	3.31
`fac_rec_#15`	0.115 789	0.121 19	4.66
`fac_rec_#20`	0.115 733	0.121 723	5.18
`fac_rec_#25`	0.115 687	0.122 259	5.68
`fac_rec_#30`	0.115 541	0.122 266	5.82

To assess the goodness of the model, since our primary goal is to have a faithful order relationship between programs, we analyse how much the differences computed between measures and between predictions differ, for each couple of programs. We can see these error values plotted in Figure 4.2.6, where the colour represents the difference between the values of the two previous heat map (i.e., $(P_y^{measured} - P_x^{measured}) - (P_y^{predicted} - P_x^{predicted})$). Again, we can see that generally the major part of the difference is made up by the function. In Figure 4.2.7 we plotted the same errors but in absolute value.

Eventually, in Figure 4.2.8 we show the order relationship as predicted by our oracle. The ratio of correctly predicted relationships is 31.27 %, a very low value. This may mean that the model is not accurate enough to appreciate the differences introduced by the ratios

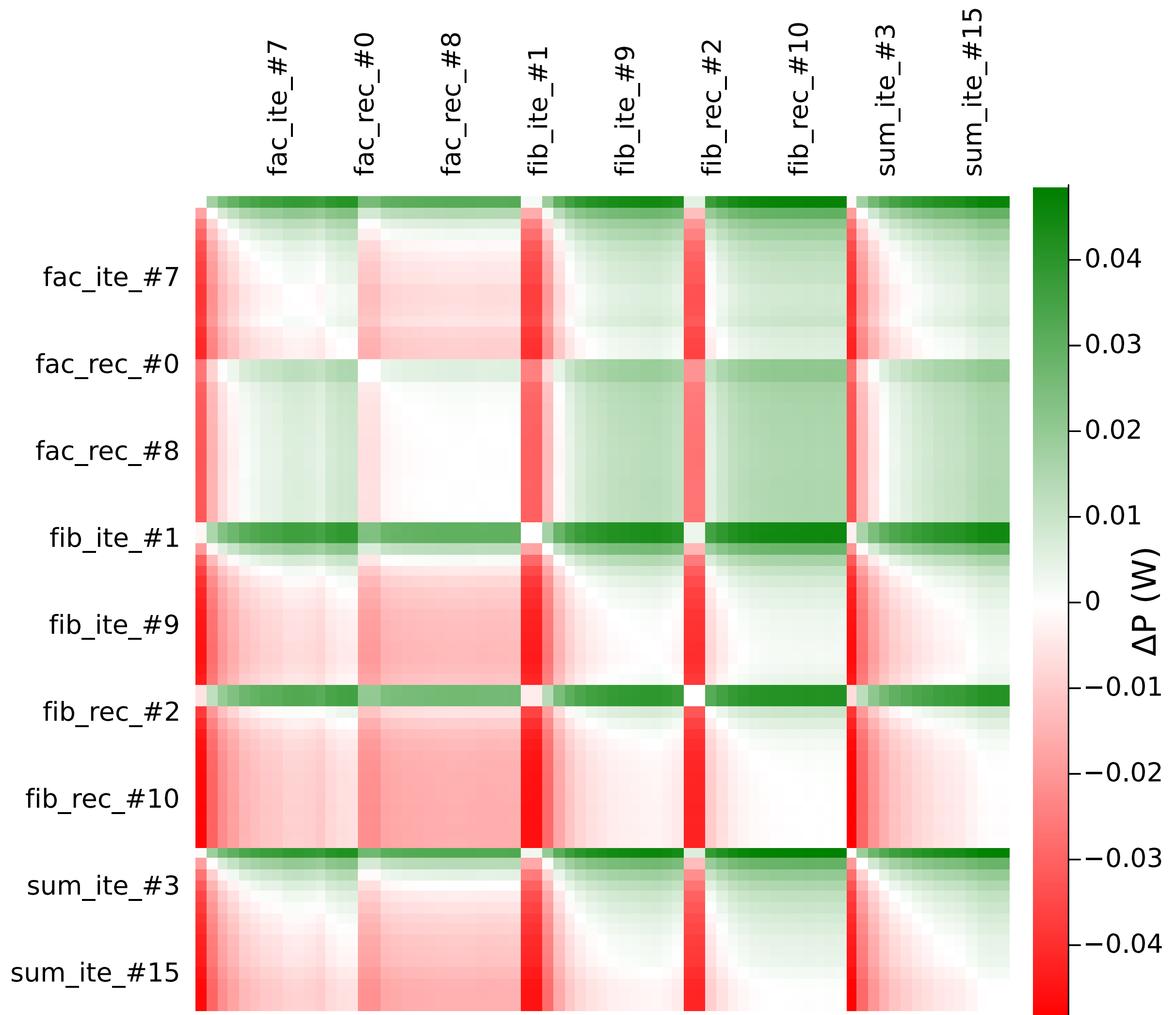


Figure 4.2.4: Difference heatmap of measured power consumptions for the deterministic programs. Each point represent the difference between the measured power consumption of the program indicated on the right and the one indicated on the top.

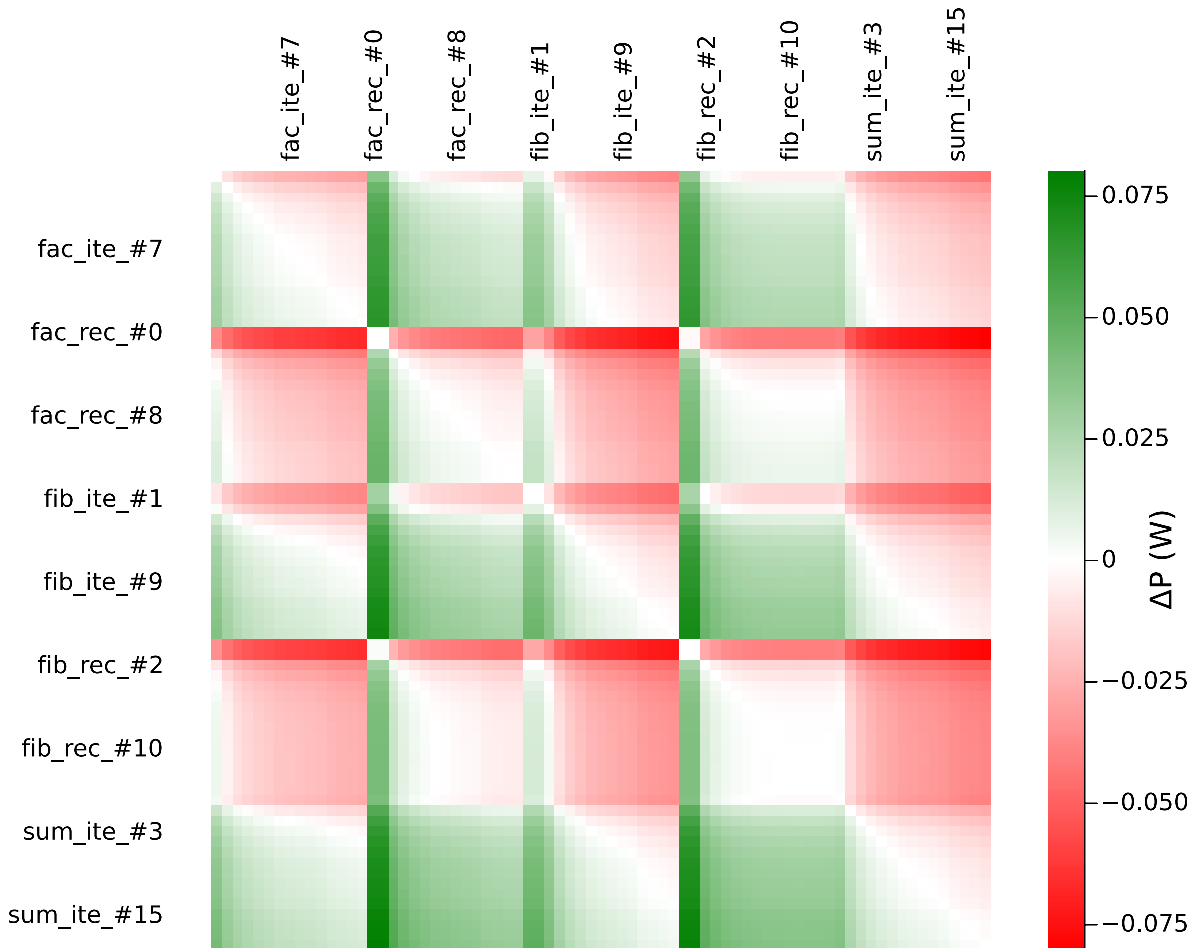


Figure 4.2.5: Difference heatmap of predicted power consumptions for the deterministic programs, measured with QEMU. Each point represent the difference between the predicted power consumption of the program indicated on the right and the one indicated on the top.

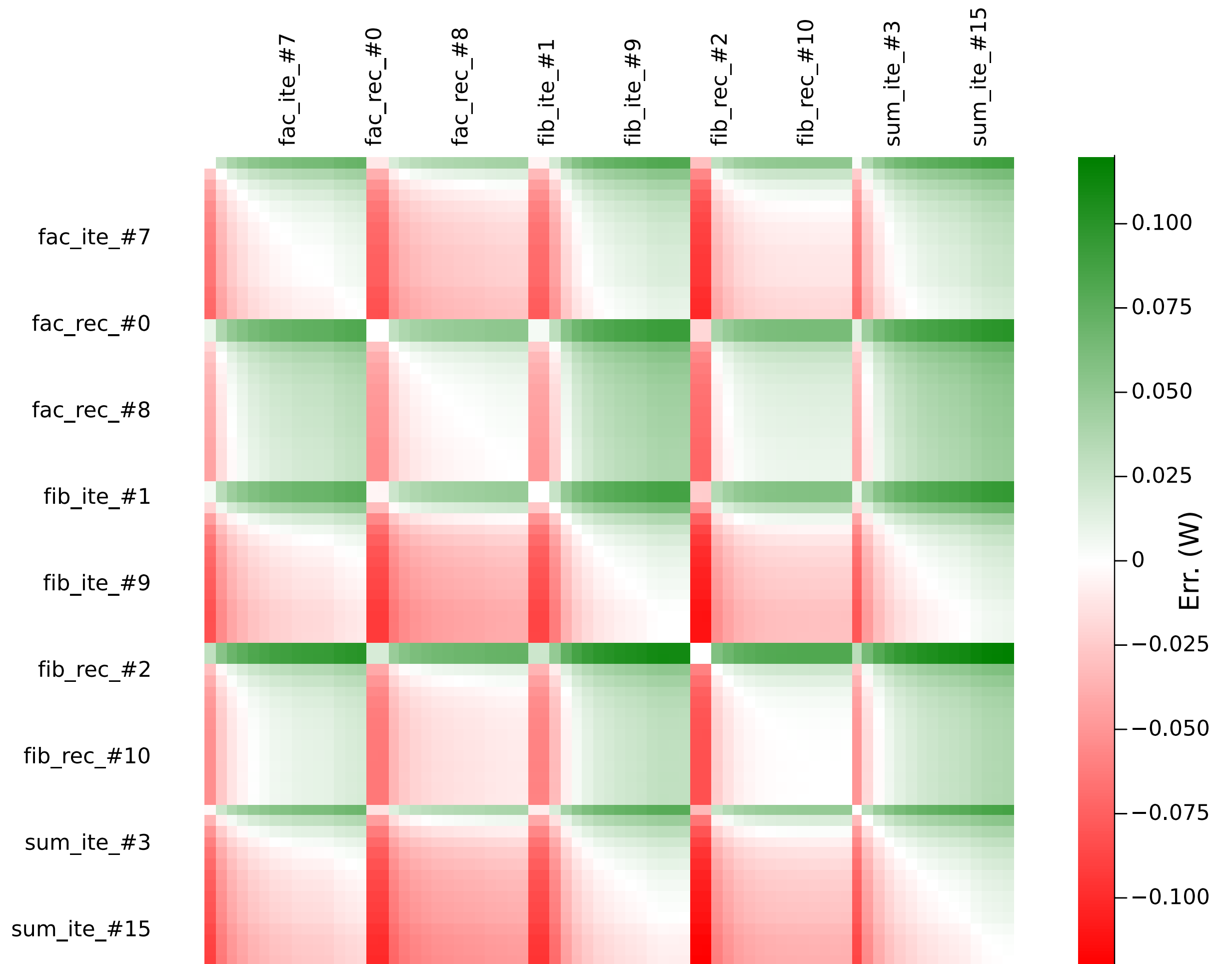


Figure 4.2.6: Error heatmap between measured and predicted power consumptions for the deterministic programs, measured with QEMU. Each point represent the error between the differences in measured and predicted power consumption of the program indicated on the right and the one indicated on the top.

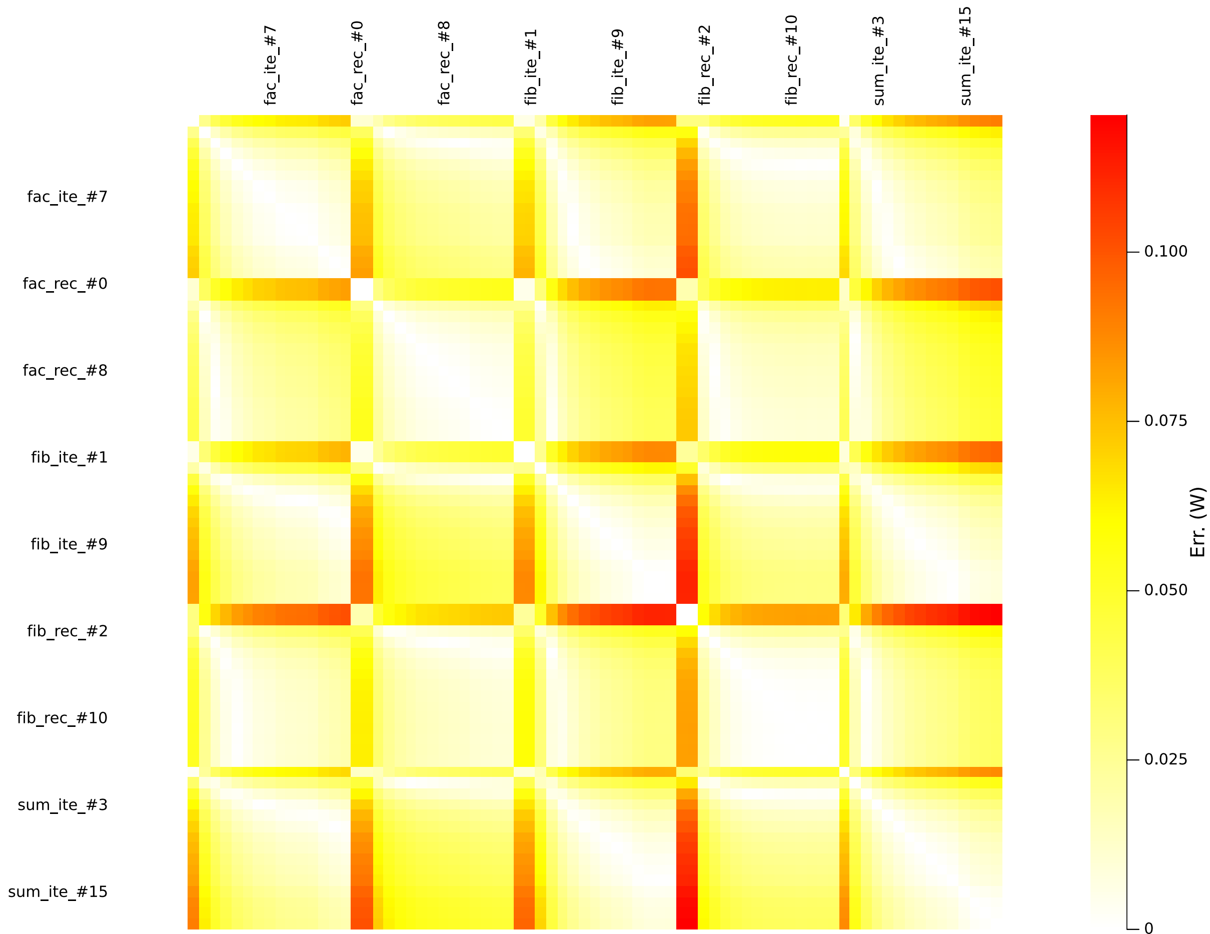


Figure 4.2.7: Error heatmap between measured and predicted power consumptions (absolute value) for the deterministic programs, measured with QEMU. Each point represent the error between the differences in measured and predicted power consumption of the program indicated on the right and the one indicated on the top.

of instructions executed (which is the effect we observed related to the argument of the functions), or that the trace provided by QEMU and GDB is not faithful to the physically executed one.

Estimating a binary program

The second type of prediction we can make is when we predict the consumption of the entire binary, without estimating the deterministic sequence of instructions. To test such a case, and at the same time checking the integrability of our model with a modified compiler, we used a battery of 58 benchmarks employing the Taffo framework [3]. Doing an analysis similar to the previous section, we see the heat maps of the measured and predicted power consumptions in Figures 4.2.9 and 4.2.10, respectively. In this case the programs are partitioned with respect to whether the optimization is active or not: benchmarks ending with the string `orig` (the first part of the matrix axes) are normally compiled; benchmarks ending with the suffix `core2_1_100_100` are compiled with the Taffo optimizations (the second part of the axes). This time the differences between the measured powers are much more pronounced, while between the predicted ones are much more attenuated.

The error heat maps are shown in Figures 4.2.11 and 4.2.12 for respectively signed and absolute error values. With respect to the deterministic programs above, these values are overall higher, with the maximum value around 30 % higher in magnitude. For a comparison, we show in Table 4.7 a selection of prediction errors from the dataset.

Figure 4.2.13 shows the sign error heat map. This time, the order relationship is preserved in more cases: the success rate increased up to 71.76 %.

Finally, we may restrict our analysis to the optimization difference for each benchmark, that is, we check if the oracle correctly predicts the difference sign between the normal and the Taffo optimized program. The results are shown in Table 4.8: the success rate is 86.21 %.

Using the binary estimate with the deterministic programs

Given the significative difference in the results, one may wonder if using the binary prediction may improve the results for the deterministic programs of Section 4.2.2 too. Doing so leads to the results in Figures 4.2.14, 4.2.15, 4.2.16 and 4.2.17. The differences between predicted powers are much reduced, as are the errors. For completeness, in Table 4.9 we show the prediction errors of the same program shown at page 62. The success rate of the order relationship improved, raising to 64.66 %.

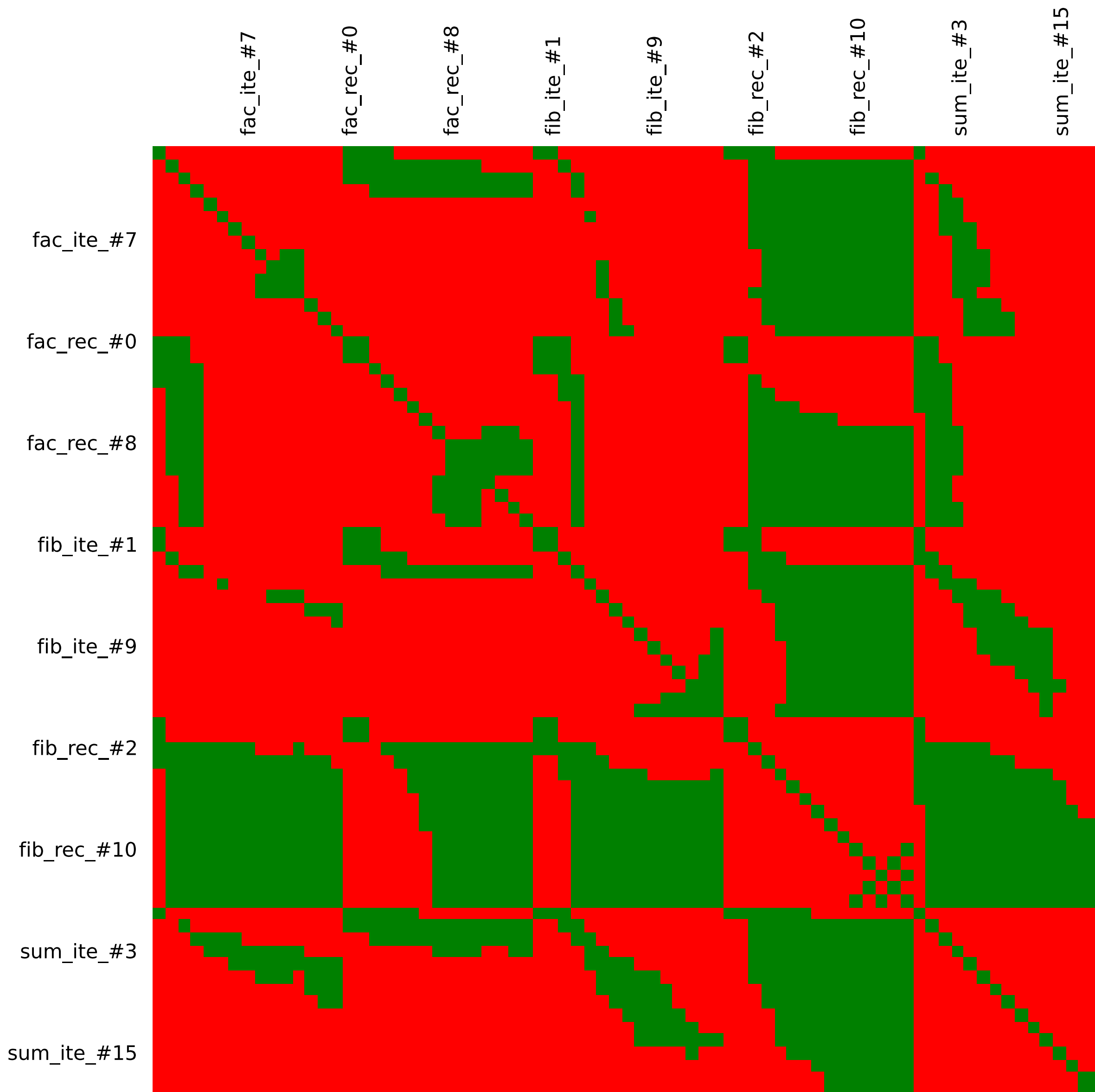


Figure 4.2.8: Sign error heatmap between measured and predicted power consumptions for the deterministic programs, measured with QEMU. A green point means that the measured and predicted differences for the program indicated on the right and the one indicated on the top have the same sign; a red one that they have opposite signs.

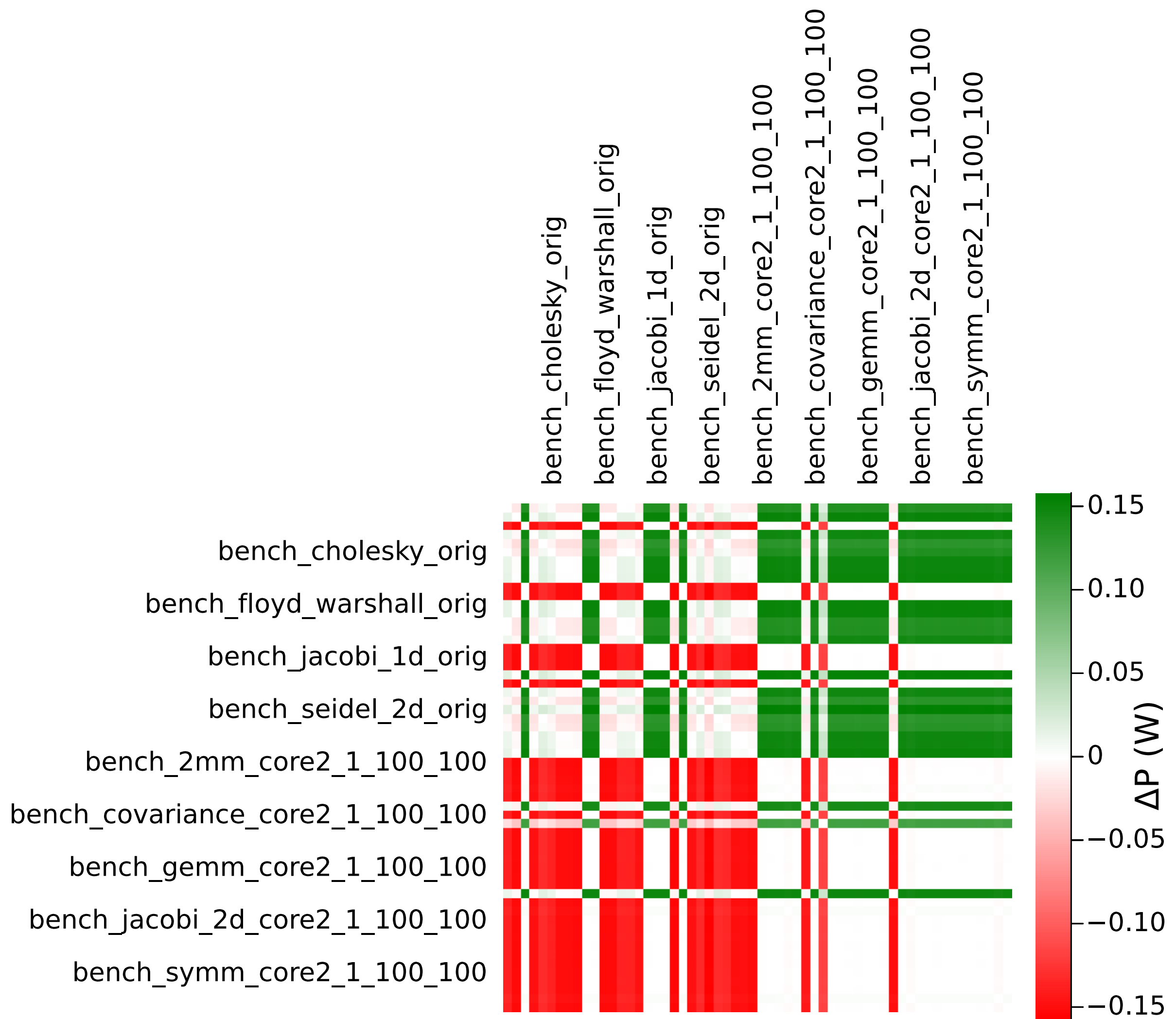


Figure 4.2.9: Difference heatmap of measured power consumptions for the Taffo benchmarks. Each point represent the difference between the measured power consumption of the program indicated on the right and the one indicated on the top.

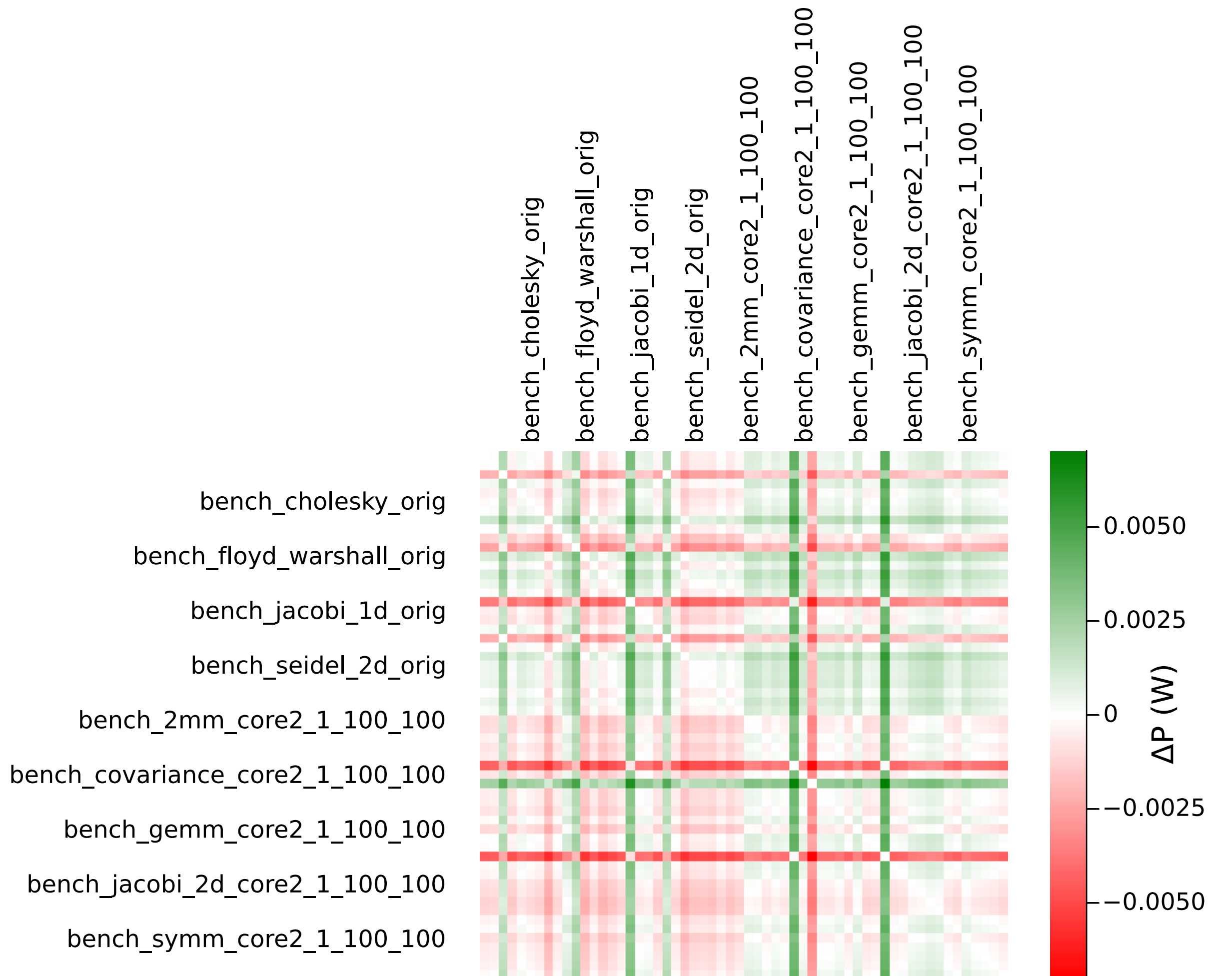


Figure 4.2.10: Difference heatmap of predicted power consumptions for the Taffo benchmarks. Each point represent the difference between the predicted power consumption of the program indicated on the right and the one indicated on the top.

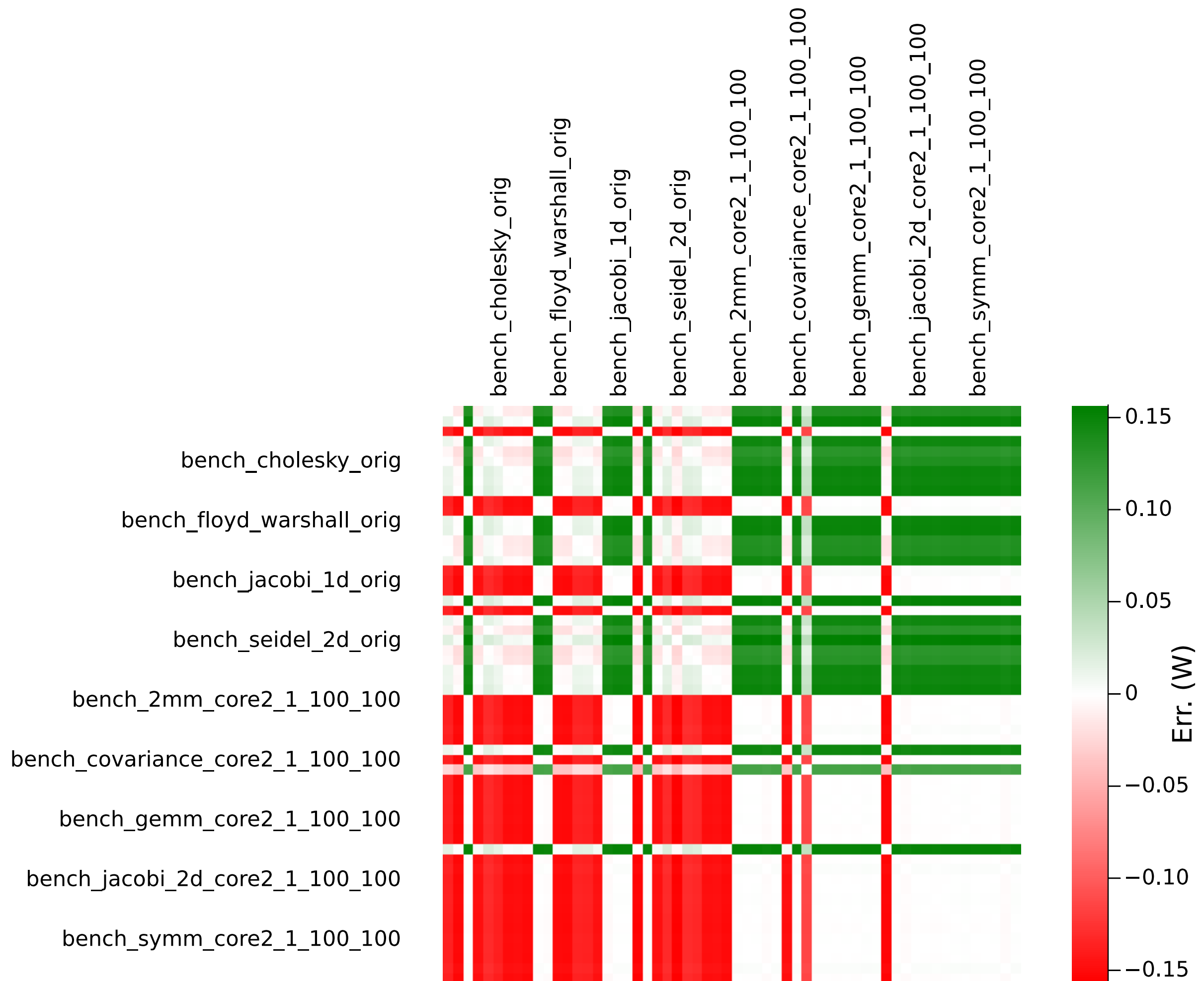


Figure 4.2.11: Error heatmap between measured and predicted power consumptions for the Taffo benchmarks. Each point represent the error between the differences in measured and predicted power consumption of the program indicated on the right and the one indicated on the top.

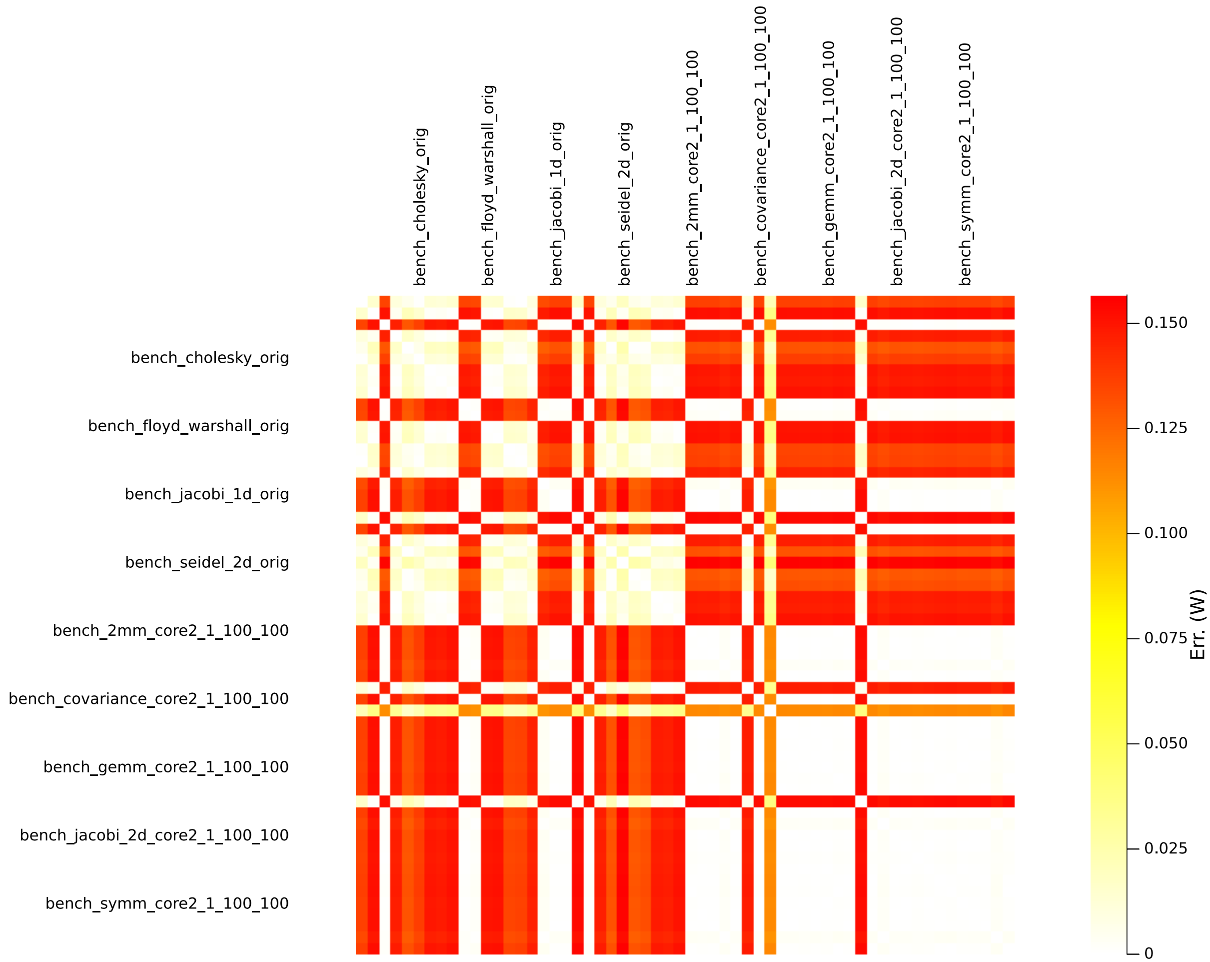


Figure 4.2.12: Absolute error heatmap between measured and predicted power consumptions for the Taffo benchmarks. Each point represent the error between the differences in measured and predicted power consumption of the program indicated on the right and the one indicated on the top.

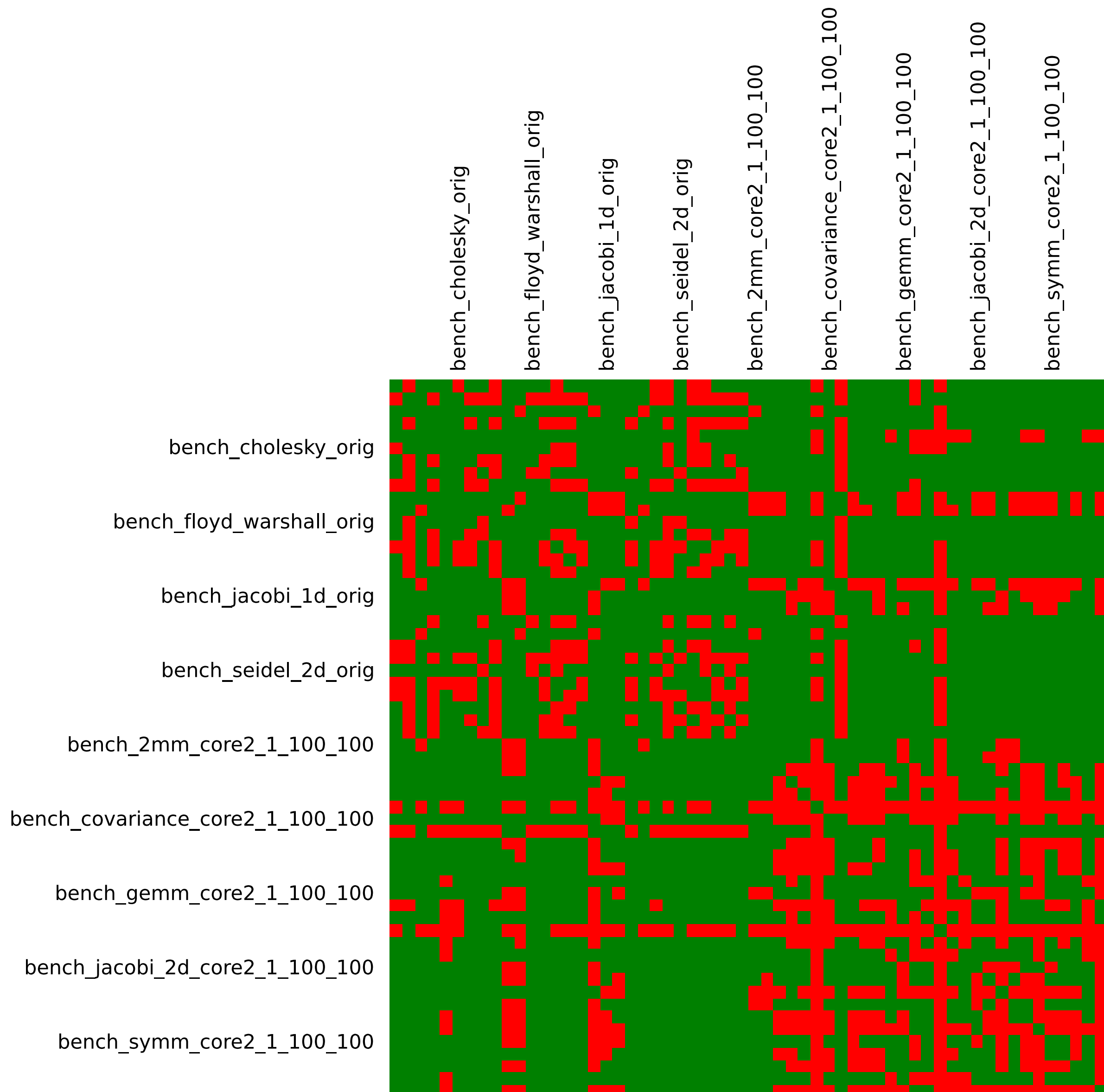


Figure 4.2.13: Sign error heatmap between measured and predicted power consumptions for the Taffo benchmarks. A green point means that the measured and predicted differences for the program indicated on the right and the one indicated on the top have the same sign; a red one that they have opposite signs.

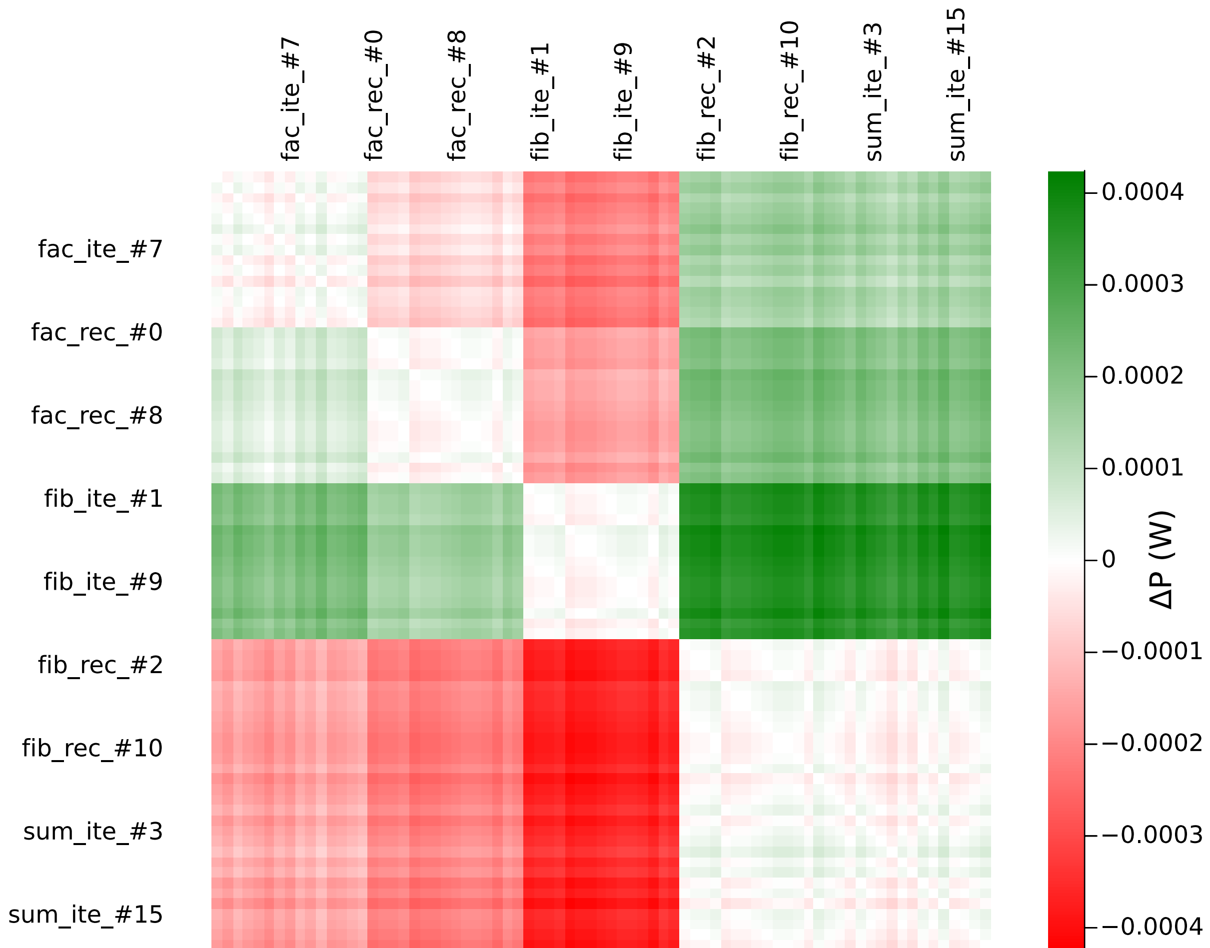


Figure 4.2.14: Difference heatmap of predicted power consumptions for the deterministic programs, measured with binary estimation. Each point represent the difference between the predicted power consumption of the program indicated on the right and the one indicated on the top.

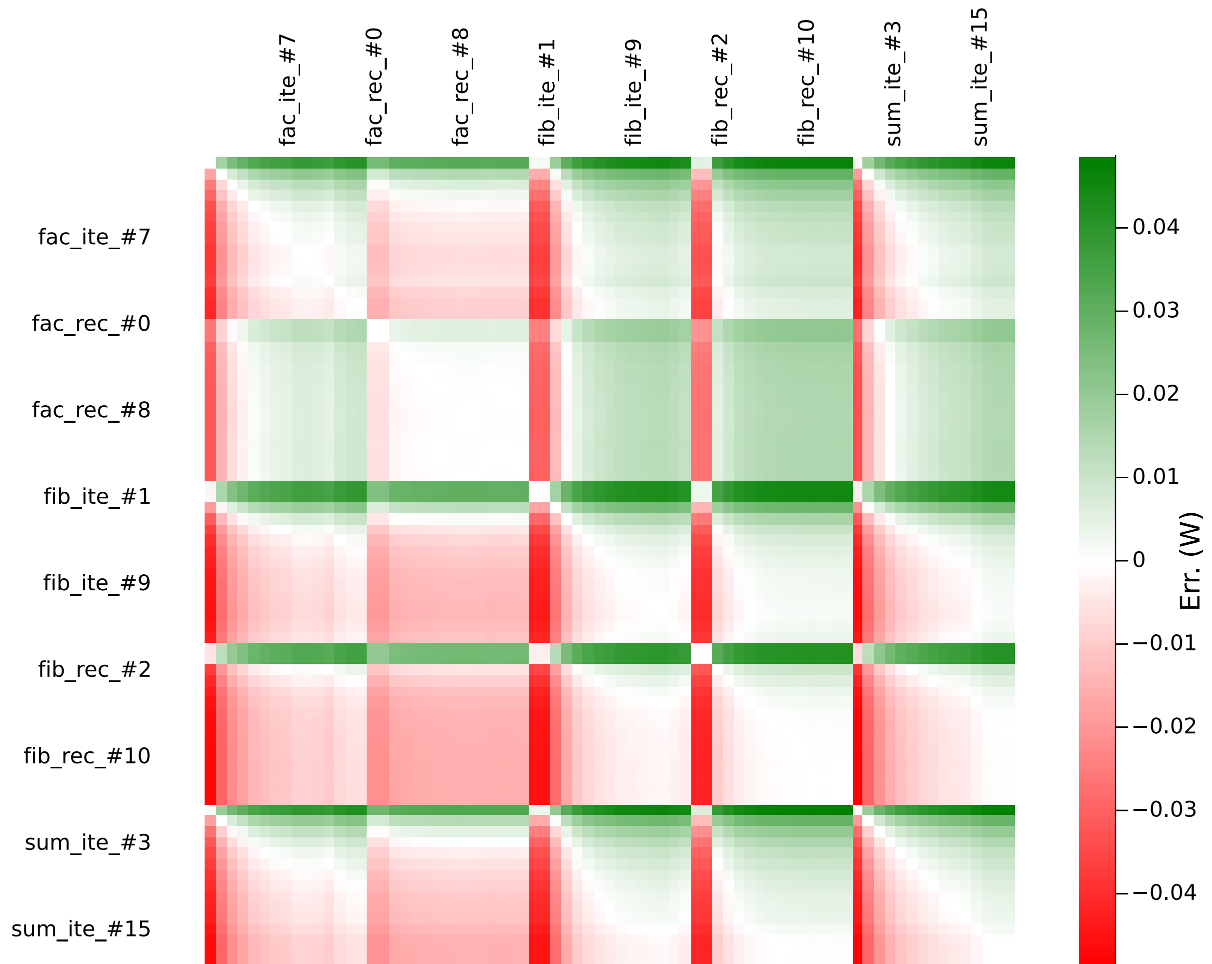


Figure 4.2.15: Error heatmap between measured and predicted power consumptions for the deterministic programs, measured with binary estimation. Each point represent the error between the differences in measured and predicted power consumption of the program indicated on the right and the one indicated on the top.

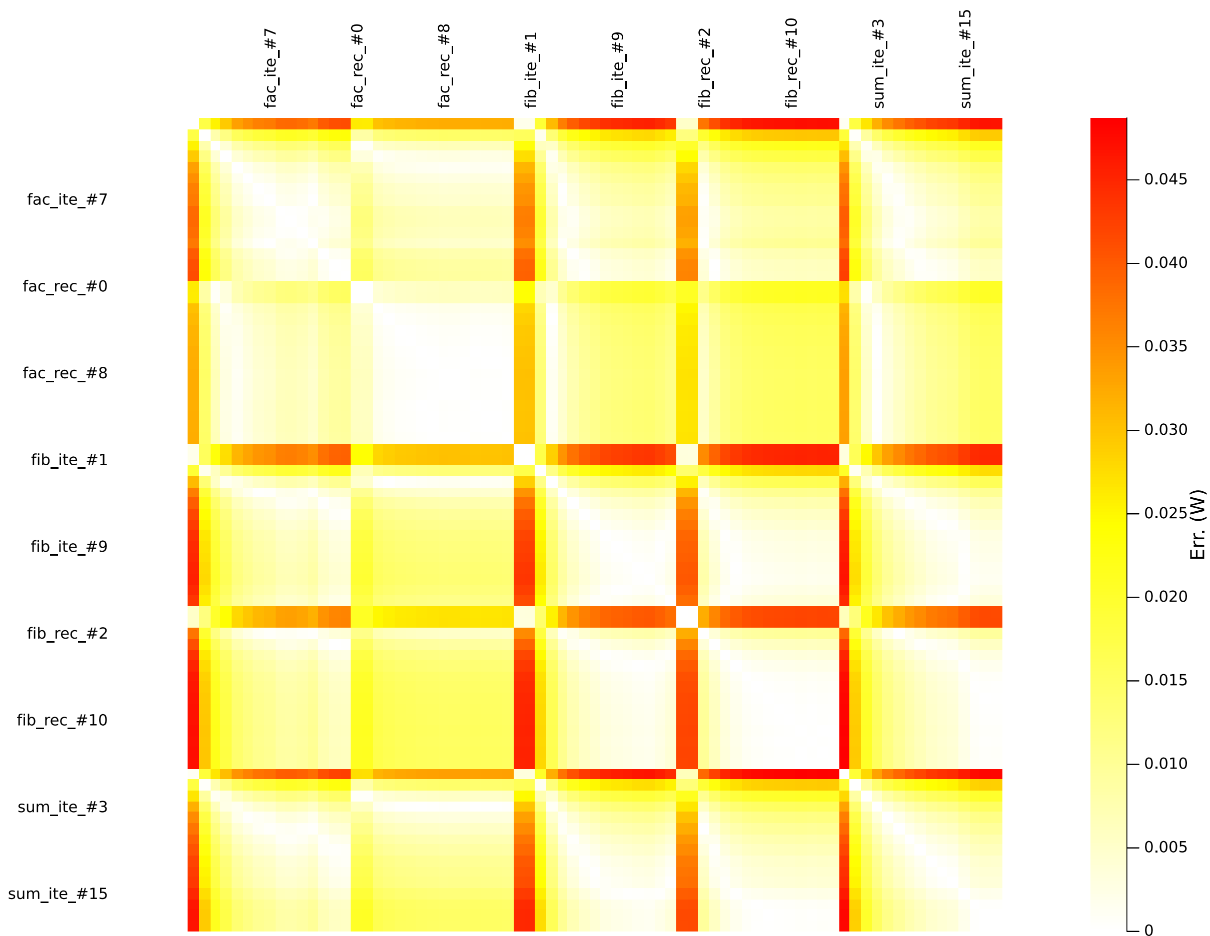


Figure 4.2.16: Error heatmap between measured and predicted power consumptions (absolute value) for the deterministic programs, measured with binary estimation. Each point represent the error between the differences in measured and predicted power consumption of the program indicated on the right and the one indicated on the top.

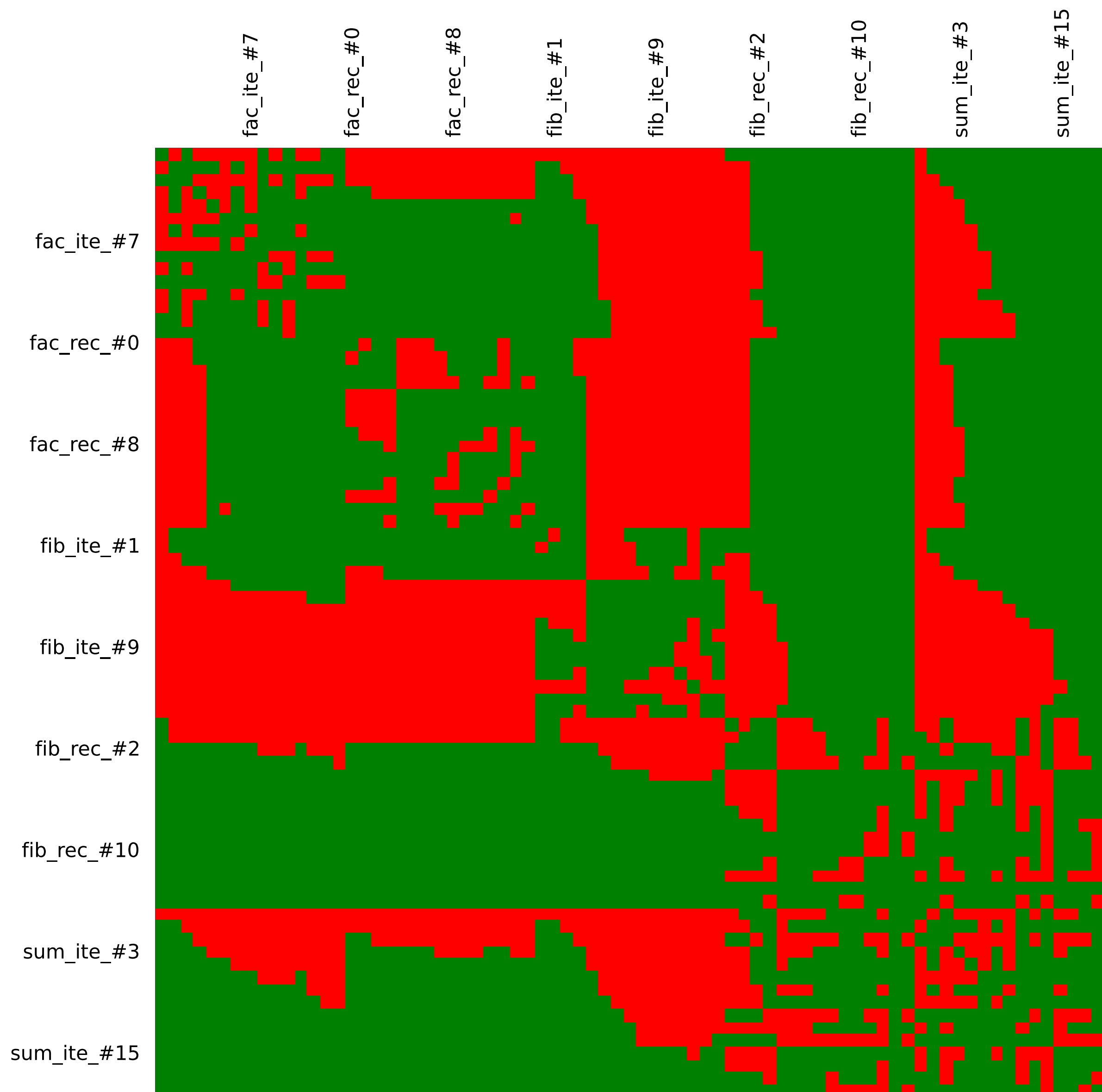


Figure 4.2.17: Sign error heatmap between measured and predicted power consumptions for the deterministic programs, measured with binary estimation. A green point means that the measured and predicted differences for the program indicated on the right and the one indicated on the top have the same sign; a red one that they have opposite signs.

Table 4.7: Results for some Taffo benchmarks.

Program (bench_)	Measured (W)	Predicted (W)	Error (%)
covariance_orig	0.240 614	0.113 281	−52.92
covariance_core...	0.090 177	0.112 405	24.65
floyd_warshall_orig	0.242 304	0.114 232	−52.86
floyd_warshall_core...	0.090 344	0.113 043	25.13
gramschmidt_orig	0.237 356	0.113 243	−52.29
gramschmidt_core...	0.239 327	0.108 636	−54.61
heat_3d_orig	0.090 164	0.109 528	21.48
heat_3d_core...	0.089 995	0.112 997	25.56
jacobi_1d_orig	0.089 865	0.112 666	25.37
jacobi_1d_core...	0.092 637	0.112 889	21.86
jacobi_2d_orig	0.089 777	0.112 503	25.31
jacobi_2d_core...	0.089 675	0.112 338	25.27

Table 4.8: Results for the Taffo optimization process. The symbol 🟢 means that the oracle correctly predicted the difference sign between the measured and the predicted powers; 🔴 means the opposite.

Benchmark program	Result	Benchmark program	Result
bench_2mm	🟢	bench_gramschmidt	🔴
bench_3mm	🟢	bench_heat_3d	🔴
bench_adi	🟢	bench_jacobi_1d	🟢
bench_atax	🟢	bench_jacobi_2d	🟢
bench_bicg	🟢	bench_lu	🟢
bench_cholesky	🔴	bench_ludcmp	🟢
bench_covariance	🟢	bench_mvt	🟢
bench_deriche	🔴	bench_nussinov	🟢
bench_doitgen	🟢	bench_seidel_2d	🟢
bench_durbin	🟢	bench_symm	🟢
bench_fdtd_2d	🟢	bench_syr2k	🟢
bench_floyd_warshall	🟢	bench_syrk	🟢
bench_gemm	🟢	bench_trisolv	🟢
bench_gemmver	🟢	bench_trmm	🟢
bench_gesummv	🟢		

Table 4.9: Results for the recursive fact function, measured with the binary estimation method.

Program	Measured (W)	Predicted (W)	Error (%)
fac_rec_#0	0.121 555	0.123 552	1.64
fac_rec_#1	0.121 587	0.123 541	1.61
fac_rec_#2	0.117 503	0.123 541	5.14
fac_rec_#3	0.116 771	0.123 53	5.79
fac_rec_#4	0.116 257	0.123 571	6.29
fac_rec_#5	0.116 123	0.123 56	6.4
fac_rec_#6	0.115 823	0.123 56	6.68
fac_rec_#7	0.115 683	0.123 549	6.8
fac_rec_#8	0.115 455	0.123 541	7.0
fac_rec_#9	0.115 474	0.123 53	6.98
fac_rec_#10	0.115 523	0.123 53	6.93
fac_rec_#15	0.115 789	0.123 537	6.69
fac_rec_#20	0.115 733	0.123 56	6.76
fac_rec_#25	0.115 687	0.123 519	6.77
fac_rec_#30	0.115 541	0.123 537	6.92

5 | Conclusion

In this work we have explored the feasibility of building a power consumption oracle for a given processor, such that it can be integrated into a compiler optimization toolchain. In doing so, we analysed a specific processor and collected some validation data for it. We can now comment the results of our model and outline the requirement to integrate such a model in a more general tool.

5.1. Model results

As we previously stated, the goodness of a linear model such as the one we used is as good as the data used to train it. Since it is unfeasible to collect data for every combination of instructions, we resorted to a multi-step prediction, where progressively less specialised model are queried. We must note however that the single models are far from perfect, and that testing as many combinations as possible is extremely important.

Moreover, we saw that a linear model may need to include non linear functions to describe the more complex parts of the processor, such as the cache management. On the other hand, using a single, so complex model for all kinds of processors may introduce overfitting, as we cannot know in advance which behaviours will the processor under test exhibit. The ideal approach would be to manually check the presence of all the dependencies we could identify, and then select just the useful model variables.

Generally speaking, the final oracle we built is able to correctly identify changes in power consumption in more than half of the cases. However, this does not necessarily mean that the model predictions will correctly guide a compiler optimizations, as they operate on different granularity levels. We can take as example cases the validation conducted on the optimizations performed by the Taffo toolchain, which gave good results; however, it is not a sufficiently exhaustive validation with respect to an actual compiler toolchain. This is left as future work.

5.2. Model feasibility

We can now summarise the difficulties of building an oracle, and how they can be mitigated. In general, we can say that our initial goal of having a general purpose oracle building method, requiring only basic input data regarding the target processor, can be reached if and only if some conditions are met.

1. The identified general model must be valid for all processors, or at least for a large range of them. Since including a new processor in the compiler toolchain should be a relatively fast operation, adapting the model to a particular CPU should be simple and quick. The general model (or the adapted particular one) should not present under nor overfitting.

2. It must be possible to collect the data in an easy and accurate way. Also, a professional setup is desirable, as it accelerates data collection and allows for better precision. The processor under test should be injectable with arbitrary code in an easy way.

3. It must be possible to access to all the information required to produce the estimate. In practice, this means doing the prediction at a stage when such information is available. For instance, since we noted the high dependency of power consumption on instruction binary weight, we needed to know the binary instruction representations that would end in memory. This meant that we needed to work at the assembly level: higher level representations that mask the register allocations, or the final ISA mapping (such as, for instance, the LLVM-IR), couldn't have predicted these effects.

The data collecting velocity of requirement number 1 may be improved by generating a test suit for different types of instructions. For instance, one may have a set of parameters (like register numbers) to be tested for three registers numeric computing instructions, like `add`, `sub`, `mul`; a set to be used for two registers instructions, like `cmp` and `mov`, and so on. Such suit may be generated by a tool, parametrized with the set of instructions and their type. This will not cover all instructions (since some of them are specific to the processor family), but it may help in speeding up part of the process.

Also, we reiterate that requirement number 2 is particularly important for the total goodness of the predictor: in our case, for instance, the memory instructions (loads, stores, `push` and `pop`) were particularly difficult to test, and collecting fewer data points may reduce the accuracy of the model. By deeply analysing the firmware code it may be possible to solve the issue, but that is not guaranteed and it would require an ad hoc analysis for each processor, partially contrasting with requirement 1.

In the end, we conclude that building such an oracle is possible, but:

- its accuracy may be limited (an exhaustive validation on an optimization toolchain has to be conducted)
- its building may be time consuming, and the portability of the data collecting method depends on a series of factors

A | Instruction generation code

To create such a stimulus suite, we developed a couple of functions to generate these sparse interval of values, and format them as correct instructions.

```
1  (defun instruction-cartesian-product (format ranges)
2    "Insert into the buffer the cartesian product of the FORMAT string.
3  RANGES is a list of ranges."
4
5    ;; check the number of args is consistent
6    (let ((contiguous-format (check-placeholders format))
7          (format-placeholders-count (count-placeholders format))
8          (ranges-count (length ranges)))
9      (when (not contiguous-format)
10       (error "Format list %s contains gaps or missing placeholders, check
   ↪  it" format))
11     (when (not (= format-placeholders-count ranges-count))
12       (error "`instruction-cartesian-product' called with %d placeholders
   ↪  but %d ranges"
13              format-placeholders-count
14              ranges-count))
15     ;; base case (no more placeholders): simply insert the string
16     (if (= format-placeholders-count 0)
17         (insert format ?\n)
18       ;; else, recursion
19       (let ((range (car ranges))
20             (other-ranges (cdr ranges)))
21         (cartesian-recurse format range other-ranges)
22         ))))
23
24 (cl-defmethod cartesian-recurse (format (obj string-range) other-ranges)
25   "Perform recursion of `instruction-cartesian-product', for the object OBJ
   ↪  of type `STRING-RANGE'.
26
```

```
27 FORMAT is the format string, OTHER-RANGES the other ranges to process."
28
29 (dolist (elt (strings obj))
30   (instruction-cartesian-product (partially-format format elt (index obj))
     ↪  other-ranges))
31 )
32
33 (defun partially-format (string object index)
34   "Format placeholders in STRING with OBJECT.
35
36 For now, it only handles placeholders marked with
37 `placeholder-marker', and objects can be ranges of strings or
38 numbers."
39
40   (save-match-data
41     (with-temp-buffer
42       (insert string)
43       (goto-char (point-min))
44       (while (re-search-forward (concat placeholder-marker
         ↪  (number-to-string index))
45                                 nil ; unbounded
46                                 t   ; simply exit when no more matches
47                                 )
48
49         (let ((object (if (stringp object)
50                           object
51                         (if (numberp object)
52                             (number-to-string object)
53                           (error "Check partially-format object type!")))))
54           (replace-match object)
55           ))
56       (buffer-string))
57     ))
```

Listing A.1: Cartesian product range code.

In Listing A.1 we can see the core of the routine, the function `instruction-cartesian-product`. It is a recursive function, that expects a format string decorated with placeholders, and a string containing a space separated list of ranges to be substituted. The default placeholder is the `%` character, followed by an integer. Placeholders with the

same number will be replaced by the same text during expansion. In the base case, a simple string with no placeholders will be passed as argument: in this case, the convenience function `count-placeholders` will return 0, the condition at line 16 will be true, and the function will simply insert the provided string in the current Emacs buffer. On the other hand, when the format function does contain some placeholders, the condition is false, and the function `cartesian-recurse` at line 21 is called. This is actually a method from EIEIO, the Emacs Lisp object system (which was inspired from the Common Lisp one), and it performs dynamic dispatch based on what is the type of the range it is called on. The ranges (not shown in this listing) were defined of two types: `numeric-range` and `string-range`. They are parsed from user provided strings: a `numeric-range` string is of the form `0:2:10`, meaning a sequence starting from 0, ending at 10, jumping by steps of 2 (if the step number is omitted it defaults to 0); a `string-range` string is a sequence of words separated by colons of arbitrary length, like `apple:pear:oranges`. For simplicity, we only show the behaviour in the simpler case of the two, that is with the `string-range`. There, the code cycles over all the words in the range with the `dolist` macro at line 29, and for each one it recurse on `instruction-cartesian-product`, but this time the format string will be first processed by `partially-format`, and the ranges will be the remaining ones, i.e. all but the one currently being processed. The function `partially-format` returns the provided string substituting the required placeholder with `object`.

The net effect is that each time there is a placeholder, `instruction-cartesian-product` will replace it with the correct element of the range using functions `cartesian-recurse` and `partially-format`, then it recurse with the reduced string and ranges.

This in the end allows us to do

```
(instruction-cartesian-product "counting %1"
                                (list (parse-range "0::10" 1)))
```

and seeing inserted into the current buffer the text

```
counting 0
counting 1
counting 2
counting 3
counting 4
counting 5
counting 6
counting 7
```

```
counting 8
counting 9
counting 10
```

A more complex example, and similar to a real ARM instruction is

```
    (instruction-cartesian-product "%1 r0, r0, #%2"
                                   (list (parse-range "add:sub" 1)
                                         (parse-range "0:10:10" 2)))
```

giving

```
add r0, r0, #0
add r0, r0, #10
sub r0, r0, #0
sub r0, r0, #10
```

B | Data analysis code

The DataFrame containing the measured data points has been constructed as shown in Listing B.1. The `metadata` function takes as argument a string containing the instruction under test, and optionally its binary encoding and its address in the executable. The instruction in first preprocessed by an auxiliary function, so to have a consistent naming convention in the table. Then, the DataFrame is constructed. The variables ending with the `_sym` suffix or starting with a colon are symbols, and they are used to index the columns of the table. The function returns a DataFrame with a single row, that will be later concatenated to the other ones.

The auxiliary functions used to decorate the data point are the following:

- `get_binary_encoding` returns the binary encoding of the instruction as a string, representing the hexadecimal binary word corresponding to it in memory. It does so by assembling the instruction on the fly
- `get_mnemonic` extracts the mnemonic opcode of the instruction (like `add`, `sub` or `nop`)
- `is_apsr` checks whether the instruction has the "`s`" flag, which sets the processor status register. There are actually two status registers in the ARM architecture, the APSR and the current program status register (CPSR), but the distinction is not relevant to the purpose of this work
- `is_conditional` checks whether the instruction uses conditional execution (like `addeq`)
- `has_barrel_shift` checks whether the instruction uses the barrel shift for its second operand, like `add r0, r1, r2, lsl #2`
- `barrel_shift_amount` returns the amount of bits to shift by when the instruction has a barrel shift. If the instruction does a shift by a register, we set this to 0, as we can't know what is the value stored in the register
- `has_immediate` checks whether the second operand is an immediate value

```julia
function metadata(instruction::AbstractString;
                  encoding::AbstractString = "",
                  address::AbstractString = "0")::DataFrame

    instruction = clean_instruction(instruction)
    if encoding == ""
        # we need to compute it
        encoding = get_binary_encoding(instruction)
    end
    try
        DataFrame(
            instruction_sym => instruction,
            :mnemonic => get_mnemonic(instruction),
            apsr_sym => is_apsr(instruction),
            conditional_sym => is_conditional(instruction),
            has_barrel_shift_sym => has_barrel_shift(instruction),
            barrel_shift_sym => barrel_shift_amount(instruction),
            has_immediate_sym => has_immediate(instruction),
            immediate_amount_sym => immediate_amount(instruction, address),
            dest_source_eq_sym => has_dest_source_eq(instruction),
            :encoding => encoding,
            binary_weight_sym => encoding_to_weight(encoding)
        )
    catch e
        println("In intruction \"$instruction\":")
        rethrow()
    end
end
```

Listing B.1: Julia code to construct the DataFrame.

- `immediate_amount` returns the immediate number of the second operand. This may be useful as it is an operand known at compile time, and as such we can perform regression on it. If the instruction does not have an immediate second operand (i.e., it is a register) the function returns 0
- `has_dest_source_eq` check whether the destination register is one of the source registers, as explained above. If this distinction is not applicable, the function returns `false`
- `encoding_to_weight` returns the binary weight of the instruction, i.e. the number of 1s in its binary encoding

C | Predictor code

C.1. Single instruction

The code is shown in Listing C.1. The first linear model tries to produce an estimate in line 7: if it fails, it throws an exception, and the control is assumed by the `catch` clause. There, the second model is tried at line 13. If even this model fails, the function assigns the constant power cost at line 20. In peculiar cases, a negative coefficient of the linear regression can assign a negative cost to an instruction, if the corresponding variable is large enough. To prevent this, we clip all power estimates to 0 W at line 25. At the end, the clock count for the instruction is extracted at line 28, and the tuple is returned.

```julia
function predict_instruction(i::DataFrame)

    _instruction = i[1, instruction_sym]
    score = 0.0

    try
        score = predict(main_model, i)[1]
    catch e
        # it means we have not trained the model for this instruction
        println("The complete model failed with instruction $_instruction")
        println(e)
        try
            score = predict(reduced_model, i)[1]
            println("Reduced: $_instruction\t$score")
        catch e
            # neither the reduced model is ok
            println("The reduced model failed with instruction
            ↪  $_instruction")
            println(e)
            println("Constant: $_instruction\t$score")
            score = mean_instruction_cost
        end
    end

    if score < 0
        score = 0.0
    end

    clock_cycles = get(instr_clock_cycles, i.mnemonic[1], 1) # default of
    ↪  1

    # return (power = score * mean_scale_factor, clock_cycles =
    ↪  clock_cycles)
    return (power = score, clock_cycles = clock_cycles)
end
```

Listing C.1: Julia code to compute the power consumption of a single instruction.

C.2. Multiple instructions

```
1  function predict_trace(trace::Vector)::AbstractFloat
2      cache = Dict{String, NamedTuple{(:power, :clock_cycles),
       ↪  Tuple{AbstractFloat, AbstractFloat}}}()
3      predictions = []
4      ## Compute score and clock cycles prediction ##
5      win_w = 2 # window width
6      for (t, n) in ((@view trace[i:i+win_w-1]) for i in
       ↪  1:length(trace)-win_w+1)
7          instruction = t[1, instruction_sym]
8          prediction = get(cache, instruction, nothing)
9          if prediction == nothing
10             prediction = predict_instruction(t)
11             cache[instruction] = prediction
12         end
13
14         instruction_next = n[1, instruction_sym]
15         prediction_next = get(cache, instruction_next, nothing)
16         if prediction_next == nothing
17             prediction_next = predict_instruction(n)
18             cache[instruction_next] = prediction_next
19         end
20
21         first_encoding = t[1, :encoding]
22         second_encoding = n[1, :encoding]
23         couple_instr_df = DataFrame(
24             avg_power_clean = mean([prediction[:power],
               ↪  prediction_next[:power]]),
25             first_mnemonic = get_mnemonic(instruction),
26             second_mnemonic = get_mnemonic(n[1, instruction_sym]),
27             first_weight = encoding_to_weight(first_encoding),
28             second_weight = encoding_to_weight(second_encoding),
29             hamming_distances = hamming(first_encoding, second_encoding)
30         )
31         scale_factor = 1.0
32         try
33             scale_factor = predict(inter_advanced_model,
               ↪  couple_instr_df)[1]
```

```
34            catch e
35                # it means we have not trained the model for this instruction
36                scale_factor = predict(inter_basic_model, couple_instr_df)[1]
37            end
38
39            if scale_factor < 0
40                println("I1: $instruction\tI2: $instruction_next\tFactor:
                  ↪  $scale_factor")
41                scale_factor = 0.0
42            end
43
44            new_power = prediction[:power] * scale_factor
45            push!(predictions, (power = new_power, clock_cycles =
              ↪  prediction[:clock_cycles]))
46        end
47
48        ## compute power score with a weighted mean ##
49        score = mean([t.power for t in predictions],
          ↪  FrequencyWeights([t.clock_cycles for t in predictions]))
50        # power_sum = 0.0
51        # cycle_sum = 0.0
52
53        return score
54    end
```

Listing C.2: Julia function to predict the power estimate of a sequence of instructions.

The code is shown in Listing C.2. We iterate through all the windows of two instructions (line 8) and we predict the estimate for the first instruction (lines 10 and 12) and for the second (lines 17 and 19). Then we construct the DataFrame with the metadata for the pair of instructions (lines 23 to 32), then we use the linear models to produce the estimate for the inter-instruction scale factor (lines 35 and 38). Here too, if the estimate is negative due to large variables with negative coefficients, we clip the scale factor to 0. We then integrate the correction by multiplying the single power estimate with the scale factor (line 40). When we have computed a prediction for all the instructions, we can integrate the clock cycles in the computation by doing a weighted mean (line 46).

Bibliography

[1] F. Bellard, "QEMU, a fast and portable dynamic translator", in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, USA: USENIX Association, Apr. 10, 2005, p. 41.

[2] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing", *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. DOI: 10.1137/141000671. [Online]. Available: https://epubs.siam.org/doi/10.1137/141000671.

[3] D. Cattaneo, M. Chiari, G. Agosta, and S. Cherubin, "TAFFO: The compiler-based precision tuner", *SoftwareX*, vol. 20, p. 101 238, Dec. 1, 2022, ISSN: 2352-7110. DOI: 10.1016/j.softx.2022.101238. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S235271102200156X.

[4] S. Chatterjee and B. Price, *Regression Analysis by Example*. John Wiley & Sons, Incorporated, 1977, ISBN: 0-471-01521-0.

[5] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization", *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 2, 15:1–15:25, Apr. 7, 2011, ISSN: 1084-4309. DOI: 10.1145/1929943.1929947. [Online]. Available: https://doi.org/10.1145/1929943.1929947.

[6] D. Elsner and J. Fenlason, *Using as*, Free Software Foundation, 2022. [Online]. Available: https://sourceware.org/binutils/docs-2.39/as.pdf.

[7] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A Comparative Study of Methods for Measurement of Energy of Computing", *Energies*, vol. 12, no. 11, p. 2204, 11 Jan. 2019, ISSN: 1996-1073. DOI: 10.3390/en12112204. [Online]. Available: https://www.mdpi.com/1996-1073/12/11/2204.

[8] K. Georgiou, S. Kerrison, Z. Chamski, and K. Eder, "Energy Transparency for Deeply Embedded Programs", *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 1, 8:1–8:26, Mar. 21, 2017, ISSN: 1544-3566. DOI: 10.1145/3046679. [Online]. Available: https://doi.org/10.1145/3046679.

[9] P. Ghiglio, “A Framework for Estimation and Visualization of Software Energy Consumption”, M.S. thesis, Politecnico di Milano, 2020.

[10] “IEEE Standard for Floating-Point Arithmetic”, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019. DOI: `10.1109/IEEESTD.2019.8766229`.

[11] “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language”, *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb. 2018. DOI: `10.1109/IEEESTD.2018.8299595`.

[12] “IEEE Standard for VHDL Language Reference Manual”, *IEEE Std 1076-2019*, pp. 1–673, Dec. 2019. DOI: `10.1109/IEEESTD.2019.8938196`.

[13] P. Landman and J. Rabaey, “Architectural power analysis: The dual bit type method”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 2, pp. 173–187, Jun. 1995, ISSN: 1557-9999. DOI: `10.1109/92.386219`. [Online]. Available: `https://dl.acm.org/doi/10.1109/92.386219`.

[14] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO ’04, USA: IEEE Computer Society, Mar. 20, 2004, p. 75, ISBN: 978-0-7695-2102-2. DOI: `10.1109/CGO.2004.1281665`. [Online]. Available: `https://dl.acm.org/doi/abs/10.5555/977395.977673`.

[15] S. Lee, A. Ermedahl, S. L. Min, and N. Chang, “An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors”, *ACM SIGPLAN Notices*, vol. 36, no. 8, pp. 1–10, Aug. 1, 2001, ISSN: 0362-1340. DOI: `10.1145/384196.384201`. [Online]. Available: `https://doi.org/10.1145/384196.384201`.

[16] B. Lewis, D. LaLiberte, R. M. Stallman, and GNU Manual Group, *GNU Emacs Lisp Reference Manual*, Free Software Foundation, 2023. [Online]. Available: `https://www.gnu.org/software/emacs/manual/pdf/elisp.pdf`.

[17] D. May, “The XMOS XS1 Architecture”, Oct. 19, 2009.

[18] D. A. Patterson, J. L. Hennessy, and P. J. Ashenden, *Computer Organization and Design, Revised Printing : The Hardware/Software Interface*. San Francisco, UNITED STATES: Elsevier Science & Technology, 2007, ISBN: 978-0-08-055033-6.

[19] M. Randolph and W. Diehl, “Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman”, *Cryptography*, vol. 4, no. 2, p. 15, 2 Jun. 2020, ISSN: 2410-387X. DOI: `10.3390/cryptography4020015`. [Online]. Available: `https://www.mdpi.com/2410-387X/4/2/15`.

[20] D. N. Reshef, Y. A. Reshef, H. K. Finucane, *et al.*, “Detecting Novel Associations in Large Datasets”, *Science (New York, N.y.)*, vol. 334, no. 6062, pp. 1518–1524, Dec. 16,

2011, ISSN: 0036-8075. DOI: 10.1126/science.1205438. pmid: 22174245. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3325791/.

[21] E. Schmidt, G. von Colln, L. Kruse, F. Theeuwen, and W. Nebel, "Memory power models for multilevel power estimation and optimization", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 2, pp. 106–109, Apr. 2002, ISSN: 1557-9999. DOI: 10.1109/92.994987.

[22] R. M. Stallman, *GNU Emacs Manual*, Free Software Foundation, 2023. [Online]. Available: https://www.gnu.org/software/emacs/manual/pdf/emacs.pdf.

[23] R. M. Stallman and GCC Developer Community, *Using the GNU Compiler Collection*, GNU Press, 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc.pdf.

[24] R. M. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB*, Free Software Foundation, 2023. [Online]. Available: https://sourceware.org/gdb/download/onlinedocs/gdb.pdf.

[25] STMicroelectronics, *STM32F405xx, STM32F407xx datasheet (DS8626), rev. 9*, STMicroelectronics, Aug. 14, 2020. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f407vg.pdf.

[26] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee, "Instruction level power analysis and optimization of software", *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 13, no. 2, pp. 223–238, Aug. 1, 1996, ISSN: 0922-5773. DOI: 10.1007/BF01130407. [Online]. Available: https://doi.org/10.1007/BF01130407.

Ringraziamenti

Non credo al mito del self-made man. Ognuno è cresciuto dalle relazioni che intreccia con chi gli sta intorno. Per questo sarebbe impossibile ringraziare tutti coloro che andrebbero ringraziati, ma ci proverò comunque. In primo luogo il professor Agosta, che ha seguito questa tesi come relatore. Sempre, davvero sempre disponibile per qualsiasi dubbio, ha monitorato il lavoro in maniera costruttiva e sempre gentile, indirizzando la ricerca. Allo stesso tempo il dottor Cattaneo, che mi ha seguito assiduamente nella parte pratica di laboratorio e raccolta dati, fornendo un aiuto tecnico sempre preciso e di fondamentale importanza. Grazie ad entrambi. Allargando appena, ci tengo a ringraziare i colleghi del DEIB, che hanno reso piacevole lavorare perfino nelle giornate più difficili. La simpatia non è affatto cosa scontata, e hanno saputo creare un clima accogliente per uno spaurito tesista come me. Quasi tutto questo lavoro di tesi è stato condotto usando software libero o open source, e ho ricevuto aiuto da persone sparse per il pianeta, spinte dal piacere di fare qualcosa di buono per gli altri: è stato meraviglioso. Pensando al di fuori dell'università, è impossibile decidere un ordine in cui ringraziare tutti. In mezzo alle difficoltà della vita, ho potuto sempre contare su persone meravigliose. Il loro supporto è stato fondamentale per questa tesi, ma io so che la loro amicizia ha toccato la mia vita in molti altri modi, diversi per ciascuno. Il risultato finale è che mi sento enormemente grato di averli accanto. Penso al gruppo degli amici del liceo, Marco, Mattia, Ilaria, Martina, Domenico, Luca, Francesco e Alberto; penso al gruppo dell'oratorio, Luca, Andrea, Marco, Chiara, Igor, Francesco, Maurizio, Graziano, Giovanni, e ai molti altri per cui non basterebbe lo spazio della pagina. Penso alla mia famiglia, in casa e allargata. A mia madre, che mi ha insegnato a vivere; a mio padre, che mi ha insegnato a pensare; a mia sorella, che mi ha insegnato a ridere. Ma anche a tutti gli altri, zii e cugini soprattutto: se leggono queste righe, sanno che mi riferisco anche a loro. Penso a tutti gli insegnanti che ho avuto nel corso della vita, a tutti gli amici, a tutte le persone che mi hanno fatto crescere, Paolo, Francesco, e tutti coloro che hanno incrociato la loro vita con la mia. Ringrazio immensamente il Signore per avermi fatto incontrare queste persone: sono davvero un dono, su cui ora rifletto per il passato, ma che so essere declinato anche al futuro.