Fridtjof Gerdsønn Eikanger

Instruction Level Power Modeling of a RISC-V System

Master's thesis
for the degree MSc in Embedded Systems, Electronics

Trondheim, July 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU

Norwegian University of Science and Technology

Master's thesis
for the degree of MSc in Embedded Systems, Electronics

Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

© 2022 Fridtjof Gerdsønn Eikanger

Abstract

The estimation of power usage in embedded software is useful for developers of both hardware and firmware. Instruction-Level Power Modeling (ILPM) is a method of estimating such power usage which is faster but less accurate than Register-Transfer Level (RTL) simulations. We show that with a collection of data-dependent features added to a standard ILPM model, the worst case error of power estimation in one two-stage pipelined RISC-V CPU is 3.52 %, with an R^2 of 0.775. The effects of adding too many features to a model is also analyzed.

Contents

Abstract **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contribution . . . 1
1.3 Structure . . . 2

2 Theory **3**
2.1 Power dissipation in digital electronics . . . 3
2.2 Instruction-level power modeling . . . 3
2.2.1 Basic ILPM . . . 4
2.2.2 Constant power modeling . . . 5
2.2.3 Data dependent ILPM . . . 5
2.2.4 System level power modeling . . . 6
2.2.5 Other attempts at power modeling . . . 6
2.3 RISC-V . . . 7
2.4 SystemC . . . 7

3 Design **9**
3.1 RTL Target . . . 9
3.1.1 Extracting a power curve . . . 10
3.2 TLM . . . 11
3.3 Features . . . 11
3.4 Calibration and prediction process . . . 12
3.4.1 Calibration process . . . 14
3.4.2 Calibration code . . . 14
3.4.3 Prediction . . . 16

4 Evaluation **17**
4.1 Benchmarks . . . 17
4.1.1 Data dependency . . . 18

5 Results **19**
5.1 Power factor vector . . . 19
5.2 Individual statistics . . . 19
5.3 Data dependency test . . . 19
5.4 Model comparisons . . . 19

6 Discussion **29**
6.1 Interpretation of the results . . . 29

6.1.1 Differences between the power domains 29
6.1.2 Adding features and over-fitting 29
6.1.3 Data dependency 30
6.1.4 Calibration code 30
6.1.5 Accuracy 31
6.2 Weaknesses 31
6.3 Future work 31
6.3.1 Further analysis 31
6.3.2 Model improvements 32
6.4 Conclusion 32

Appendices

A Instruction timings 33

B Benchmarks 37

C Statistics 41

Bibliography 49

List of Tables

2.1 A selection of RISC-V extensions. Compiled from information in [Uni19]. . . 7

3.1 Implemented instruction-level power modeling (ILPM) feature types. 12
3.2 The instruction types of different instructions as counted by the instruction swap features. 12

4.1 Benchmarks used to evaluate the model. 18

5.1 Average power by domain for different all benchmarks extracted from register-transfer level (RTL) simulation. Normalized within each domain to the average power of the calibration code. 23
5.2 The accuracy of the instructing timing for different benchmarks. Found by dividing the execution time predicted by the model by the real execution time from the register-transfer level (RTL) simulation. 23
5.3 R^2 for all model across all domains. This R^2 is based on the ability of the models to predict average power over a program's execution. 27

A.1 Instruction completion times for all modeled instructions. A 0-aligned address is one which ends in 0 or 8. 4-aligned addresses end in 4 or c. One cycle is 62.5 ns. 33

C.1 Statistics for the calibration code benchmark. 42
C.2 Statistics for the coremark benchmark. 43
C.3 Statistics for the fib benchmark. 44
C.4 Statistics for the fir benchmark. 45
C.5 Statistics for the hanoi benchmark. 46
C.6 Statistics for the quicksort benchmark. 47
C.7 Statistics for the tak benchmark. 48

List of Figures

2.1 Shape of the ILPM vectors and matrices. 4

3.1 Shape of the multidomain ILPM matrices. 9
3.2 A diagram showing the rough layout of the TC4 compute domain. 10
3.3 The process of calibrating and utilizing the model. 13

5.1 The power factor vector for the central processing unit (CPU) module of the *basic ilpm* model. Normalized to the energy cost of delta time (10 ns). 20
5.2 The power factor vector for the central processing unit (CPU) module of the *branch taken* model. Normalized to the energy cost of delta time (10 ns). . . . 21
5.3 The power factor vector for the central processing unit (CPU) module of the *all features* model. Normalized to the energy cost of delta time (10 ns). 22
5.4 The real vs predicted power used by the CPU when running the calibration code. The model used for prediction is *single instructions*. The power has bee normalized to the average of the real power usage. 24
5.5 Real (black) vs predicted (red, dotted) power used by the domains when running different benchmarks. y-axis shows power, x-axis shows time. The *all features* model was used for all predictions. 25
5.6 Real vs predicted power used by the domains when running the data dependency test. The *all features* model was used for predictions. 26
5.7 Relative error by actual power in the different domains. The actual power is normalized to the calibration code average power. 28

Source code

3.1 calibration_code.S loop structure . . . 14
3.2 calibration_code.S loop structure . . . 15
B.1 fir.c . . . 37
B.2 fir.c . . . 37
B.3 hanoi.c . . . 38
B.4 quicksort.c . . . 38
B.5 tak.c . . . 39

Acronyms

ALU arithmetic logic unit

CMOS complementary metal-oxide-semiconductor

CPU central processing unit

CSV comma separated values

DVFS dynamic voltage and frequency scaling

I/O input/output

ILPM instruction-level power modeling

IPC instructions per cycle

ISA instruction set architecture

LSU load/store unit

PFV power factor vector

RMS root-mean-Square

RTL register-transfer level

SLPM system-level power modeling

SoC system on a chip

TLM transaction level modeling

UART universal asynchronous receiver-transmitter

Chapter 1

Introduction

1.1 Motivation

The minimization of power and energy usage in computing has been recognized as a critical component of electronic system design for over 30 years [CSB92]. Estimating the power or energy usage of a specific program in a timely manner is essential to providing embedded firmware developers the feedback they need to make energy-optimal architectural and small-scale decisions. Using a register-transfer level simulation can give accurate estimations of power usage, but these methods have several drawbacks.

First, the simulation and power analysis are prohibitively slow. Second, it requires a model of the hardware at the level of the simulation, which is unlikely to be available to developers outside the company developing the hardware. Lastly, the output of the simulation gives the developer no indication of what parts of the program consume power, just which parts or modules of the chip. The developer will then have to work backwards to find out which modules are utilized by what parts of the code.

These drawbacks justifies the utilization of a higher level simulation which runs faster and is focused more on functionality than hardware implementation. Transaction level modeling (TLM) fulfills these requirements. There is no way to directly estimate power with such a model, since the power usage is implementation-dependent. However, the model could report the occurrence of different architectural states, e.g. how many instances of each instruction is executed in a central processing unit (CPU) core, in a system trace. We wish to use this system trace to estimate the power usage of a core, by finding the correlation between it and the power usage reported by a cell-level simulation. This approach is called instruction-level power modeling (ILPM) [TMW94].

However, there is a limit to the accuracy that can be achieved by only looking at instructions executed. While two instances of an add-instruction are counted as the same by a basic ILPM model, they can have wildly different effects on the power usage based on the operands [ACPS01]. We therefore need a model which takes into account some features of the data being processed.

1.2 Contribution

In this paper, we propose several instruction-level power models with different features, and investigate whether adding features to a model will make it more accurate.

1.3 Structure

Chapter 2 will give an introduction to the theory of power dissipation in digital circuits, and a summary of previous works in the field of power modeling. Sections 2.1 and 2.3 are reworked sections from the project thesis [Eik21] which this thesis builds on.

Chapter 3 covers the design of our particular power model, including the calibration code.

Chapter 4 shows how the power model was evaluated, which statistics were collected and which benchmarks were used.

Chapter 5 contains the results of the evaluation.

Chapter 6 is a discussion of the results, the weaknesses of this paper, and future work.

Chapter 2

Theory

2.1 Power dissipation in digital electronics

Modern digital electronics is dominated by complementary metal-oxide-semiconductor (CMOS) logic, so to understand power dissipation in modern electronics we must first understand power dissipation in CMOS circuits. The three main sources of power dissipation in CMOS logic circuits are transistor switching, direct-path short circuits and leakage currents [CSB92]. For most circuits, these sources can be modeled as the three terms in equation 2.1.

$$\begin{aligned} P_{\text{total}} &= P_{\text{switching}} + P_{\text{short-circuit}} + P_{\text{leakage}} \\ &= p_t(C_L V_{\text{dd}}^2 f_{\text{clk}}) + I_{\text{sc}} V_{\text{dd}} + I_{\text{leakage}} V_{\text{dd}} \end{aligned} \tag{2.1}$$

As one would expect, all terms scale with the supply voltage. The switching power also scales with the loading capacitance C_L, the clock frequency f_{clk} and the switching activity p_t. The switching activity is a measure of how many times the average transistor flips in one clock cycle. While it is usually a number between zero and one, a single transistor can switch multiple times in one clock cycle due to glitching. $P_{\text{switching}}$ used to be dominant, but as gates and transistors get smaller, leakage current grows more important [Tho98].

Assuming no low-power states (such as sleep modes) or dynamic voltage and frequency scaling (DVFS), any firmware that runs on a core will only affect the switching activity p_t. The other variables in equation 2.1 are decided solely by the design of the circuit.

If we allow for low-power states however, the firmware will also be able to affect the leakage current by e.g. power gating parts of the circuit. If we go further and allow DVFS, V_{dd} and f_{clk} can be either directly controlled by firmware [WWDS94] (usually through an operating system), or indirectly affected by firmware as the hardware decides what voltage and frequency is appropriate [IZZH19].

2.2 Instruction-level power modeling

Instruction-level power modeling is a method of predicting the power or energy usage of a CPU, core, micro-controller, or other device containing a CPU. The basic idea is that each instruction in the CPU's instruction set architecture (ISA) can be assigned a base energy cost, representing how much energy is needed to execute the instruction on average. This cost is then used to predict the device's power usage over time, based on a trace of instructions executed by the CPU when running a specified program in a functionally correct simulation. It is possible to assign energy costs to any event in the simulation that runs the program,

$$S = \begin{pmatrix} 10 & 16 & 0 & 0 & \cdots & 0 \\ 10 & 0 & 16 & 0 & \cdots & 0 \\ 10 & 0 & 0 & 16 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 10 & 0 & 0 & 0 & \cdots & 16 \end{pmatrix}$$

(a) The system trace S of a program running each instruction 16 times. The left-most column represents the length of time interval, while the rest represent single instructions.

$$P = \begin{pmatrix} 42 \\ 26 \\ 18 \\ \vdots \\ 10 \end{pmatrix}$$

(b) Power curve P. Its rows correspond to the rows of S.

$$\alpha = \begin{pmatrix} 1 \\ 2 \\ 1 \\ 0.5 \\ \vdots \\ 0 \end{pmatrix}$$

(c) Power factor vector α. Its rows correspond to the columns of S.

Figure 2.1: Shape of the ILPM vectors and matrices.

not just which instructions are executed. We will call the events tracked by the simulation and placed in the system trace *features*.

There are three important quantities that must be explained to understand how an ILPM model is created. Shown in figure 2.1, they are the system trace S, the power curve P and the power factor vector α. P is a vector containing the power over time of a program, where each of the N elements is the average power during a small interval of time. It is obtained by running the program on a physical chip or in a power-aware low-level (e.g. register-transfer level (RTL)) simulation. The system trace S is an $N \times M$ matrix, where each column is a feature, and each row shows how many times each feature occurred during an interval of time corresponding to an element of P. The power factor vector is a length M vector, where each element is the energy cost of one feature. The PFV could be viewed as the power model itself, as it is what stays constant between different programs.

To extract the PFV α from a system trace S and power curve P, equation 2.2 is solved using a least squares error algorithm.

$$P = S\alpha \tag{2.2}$$

Some ILPM models, like [Vil19], choose to solve impose the restriction that all elements of α must be non-negative, with the justification that it does not make sense in a physical system to save energy by executing an instruction. To find the predicted power curve $\hat{P}$ of a program from a system trace S and PFV α, equation 2.3 is used.

$$\hat{P} = S\alpha \tag{2.3}$$

2.2.1 Basic ILPM

The original ILPM paper [TMW94] examines a model with two kinds of features: single instructions and inter-instruction effect features. Single instructions are as described above: the simulator tracks how many times each instruction is executed. Inter-instruction effects are the power usage resulting from how instructions executed after one another interact, and inter-instruction effect features are features designed to predict such effects. The paper divides inter-instruction effects into two categories: circuit state effects, which is the overhead between executing different instructions consecutively; and resource constraint effects, such as stalls and cache misses. Circuit state effects are found to have small enough variation to

model as a constant power contribution. The authors model the effect of cache misses as a constant multiplied by the number of misses, which is found using a cache simulator.

The original paper does not perform the estimation using an TLM processor simulator, but instead divides the program into blocks which can be statically analyzed to extract power used when executing the block once. Then, a program profiler is used to tally the number of times each block is executed. Then, the power contribution of the cache is added.

[ACPS01] describes how ILPM predicts a program's total energy cost E_p with the equation

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \tag{2.4}$$

where B_i, O_i and E_k are the costs of single instruction i, the inter-instruction cost of instruction sequence $i,\ j$, and the cost of event k respectively. Events include stalls, cache misses and similar. N_i is the number of times instruction i was executed and $N_{i,j}$ is the number of times the sequence $i,\ j$ was executed.

There are other, more granular ways to handle circuit state overhead cost prediction. [NKL+03] extracts one cost per possible instruction pair, which provides the possibility of higher accuracy, but adds $m(m-1)$ features to the model, where m is the number of instructions in the ISA. The paper reached an accuracy of 5 % on a 3 stage CPU, for various "real software kernel". [Vil19] avoids the combinatorial explosion by partitioning the instructions by encoding format into 8 groups, reducing the number of features added from over 1000 to 56. This approach reached an accuracy of 1.2 % for a RISC-V core and 2 % for an ARM core, however running only a single, undisclosed benchmark.

We will refer to a feature describing the occurrence of two instructions or instruction groups being executed consecutively as a *instruction swap*. So executing a sequence of three instructions, A, B, and C, will increment five features: the single instruction features N_A, N_B and N_C, and the instruction swaps $N_{A,B}$ and $N_{B,C}$.

2.2.2 Constant power modeling

It was shown in [RJ98] that the power of embedded high performance 32-bit CPUs can be modeled as a constant to an accuracy of 8 %. The paper argues that optimizing for program execution time will also optimize for program energy cost in almost all cases, and that the most power saving comes from physical changes like operating voltage and frequency. While voltage and frequency is undeniably a major factor in power usage, this paper was written in 1998, and power saving in processors is a lot more complex 25 years later. As we will see, many models outperform the 8 % result as well.

2.2.3 Data dependent ILPM

There is a limit to how accurate a power model can be if it is constrained to only considering which instructions are executed. Such a power model can, for example, not tell the difference between a program where two registers containing the value 0 are repeatedly added together, and one where random register values are added together. In the latter case, the arithmetic logic unit (ALU) will have higher switching activity, leading to higher power usage. Data dependent ILPM, or ILPM with data dependency, aims to increase the accuracy of the power model by including features related to the data being processed by the executed instruction.

In [ACPS01] a data-dependent model was proposed, which provides power usage cycle by cycle. In addition to single instruction and instruction swap features, it uses the hamming distance between consecutive values of buses and registers to predict extra power usage. The model also differentiates between the direction of transition, i.e. a bit flipping 0 to 1 is

different than one flipping 1 to 0. This differentiation only makes sense in a model which provides power usage cycle by cycle, as over longer periods of time the number of transitions 0 to 1 must approximately equal the number of transitions 1 to 0. We will ignore the bit flip directions. The data-dependent part of the model is described with the equation

$$\begin{aligned} f(data) =& \text{Hamming}(X_{i-1}, X_i) \times \text{weight}_X \\ &+ \text{Hamming}(Y_{i-1}, Y_i) \times \text{weight}_Y \\ &+ \text{Hamming}(\text{memaddr}_{i-1}, \text{memaddr}_i) \times \text{weight}_{\text{memaddr}} \end{aligned} \tag{2.5}$$

where Hamming is the hamming distance between two values of a signal, i.e. the number of bits that are different between the values. X_i and Y_i represent the values during cycle i of the two CPU buses which read their values from the registers and input values into the ALU and other CPU modules. The change in values on these two buses is used as a proxy for general switching activity in the ALU. Memaddr_i is the address on the memory bus, used to fetch data and instructions from memory, during cycle i.

This model achieves an error of less than 5 % for all benchmarks run.

2.2.4 System level power modeling

So far, the focus has been on power usage of CPU cores, as this is where the instructions are executed. However, in embedded system design, CPU power is sometimes not the main factor in power usage. System-level power modeling aims to describe an entire system on a chip (SoC) in terms of its component modules and their power usage.

[GVH00] shows a framework for modeling different parts of an SoC in order to help hardware developers with architectural choices early on in development. The main idea is to model individual hardware modules using ILPM, where the term instruction is used more broadly to fit modules other than a CPU. For example, a universal asynchronous receiver-transmitter (UART) was modeled using the four instructions, "reset", "enable", "disable" and "write buffer". The authors show that this model predicts power usage with up to 30 % error, for eight parametric permutations of the hardware.

In [SWN+20], the power model FUSED is used to simulate and predict the power of an intermittent computing system where memory access contributes significantly more to energy usage than which instructions are executed. The prediction is made harder by a fluctuating supply voltage which affects current draw, and the need to predict somewhat instantaneous power usage, as the goal is to predict how much calculation an intermittently powered system can perform before having to power off. Since the system is powered from a capacitor, current voltage scales with the integral of current, and so the model attempts to predict current usage, not power directly. The FUSED model predicts the average current i_{cc} over a short time interval Δt as

$$i_{cc} = \Big(\frac{\sum_k E_k c_k}{v_{\text{core}} \Delta t} + \sum_m I_{m,s}\Big) c_{\text{reg}}(V) \tag{2.6}$$

where E_k and c_k are the energy cost and number of occurrences respectively of event k, $I_{m,s}$ is the current draw of module m in the state s it is in during the time interval, and $c_{\text{reg}}(V)$ is a voltage-dependent scaling factor determined experimentally. FUSED achieves a maximum error of 23 %, with the greatest errors on tests involving memcpy.

2.2.5 Other attempts at power modeling

[Wan17] greatly simplifies the single instruction part of ILPM, dividing the instructions in the ISA by functionality into three groups: load, store and ALU. Using these three features

Table 2.1: A selection of RISC-V extensions. Compiled from information in [Uni19].

Extension	Description
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
Zicsr	Control and Status Register (CSR) Instructions
Zifencei	Instruction-Fetch Fence
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Q	Standard Extension for Quad-Precision Floating-Point
C	Standard Extension for Compressed instructions

and the instructions per cycle (IPC) of the program, the model achieves a maximum error of 10 %.

[DLL+17] uses a neural network to predict power usage of coarse-grained reconfigurable architectures with an error less than 5 %. The power usage in a CGRA is hard to estimate with a regression model, since its configuration and datapath can change during runtime, affecting the power consumption.

2.3 RISC-V

RISC-V is a free and open ISA originally created at the University of California, Berkeley in 2010 [Int21]. It is made to be flexible and applicable to many use cases. As such, it consists of several base integer ISAs, all of which have several extensions. The base ISAs differ in how many general purpose CPU registers are in each core (16 or 32), and how wide they are (32, 64 or 128 bits). The extensions add additional functionality like support for multiplication, floating point arithmetic, or atomic operations. The name of a specific RISC-V ISA with specified extensions is RV<bits><reg><extensions>, where bits is the size of the registers, base is either I (for 32 registers) or E (for 16), and extensions is a series of extensions (listed in table 2.1). For example, RV32IMF is the RISC-V ISA with 32 32-bit registers and the multiplication and floating point (single precision) extensions [Uni19].

In this paper, we are exclusively working with the unprivileged RISC-V instructions, which are those which are "generally usable in all privilege modes"[Uni19].

The RV32I instruction set contains 40 different instructions. The functionality covers basic integer- and bit-wise arithmetic, branches and jumps, program counter arithmetic, memory input/output (I/O), and system calls. This is enough to emulate the other extensions except for the Zi- and Atomicity extensions.

2.4 SystemC

SystemC is a library for C++ which enables high level description of hardware. It can be used to create virtual prototypes of hardware, which allows developers to experiment with HW/SW partitioning and other architectural choices [Ini22a]. A virtual prototype can also be used as a model of existing hardware, which can be used for simulation and analysis, e.g. a functionally correct CPU core for a specific ISA can be modeled to verify the correctness of software if the hardware is unavailable.

SystemC contains libraries for specific design methodologies like TLM, allowing developers to model interconnects between modules and create entire SoCs, and quickly simulate multiple cores and systems simultaneously [Ini22b].

Chapter 3

Design

Our design predicts power usage in multiple sub-modules (power domains) by simulating a TLM model of a RISC-V core, resulting in a system trace which is used alongside a PFV to predict a power curve. Unlike in section 2.2 the PFV is a matrix with several columns, one for each power domain. This results in a power curve which is also a matrix with several columns, one for each domain. For the sake of continuity, we will keep referring to these components as the power factor vector and power curve, despite them both being matrices.

3.1 RTL Target

The RTL target is a proprietary test chip developed by Nordic Semiconductor, known simply as *test chip 4* or *TC4*. This chip has a domain for computation containing four RISC-V cores, some memory and interconnect. This compute domain will be the target of our predictions. A diagram of its layout is shown in figure 3.2.

The cores in the compute domain are proprietary versions of the open source project nanoRV[1] which implements the RV32I ISA. The cores are 16 MHz, two-stage pipelined cores with one fetch and one execute stage. In our analysis, instruction and data caching has been disabled, and only one core (CPU 0) is active.

The domain's memory is split in two: one ultra low power and one ultra low leakage memory module.

[1] Hosted at `https://github.com/rbarzic/nanorv32`

$$S = \begin{pmatrix} 10 & 16 & 0 & 0 & \cdots & 0 \\ 10 & 0 & 16 & 0 & \cdots & 0 \\ 10 & 0 & 0 & 16 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 10 & 0 & 0 & 0 & \cdots & 16 \end{pmatrix}$$

(a) The system trace S of a program activating each feature (columns) 16 times.

$$P = \begin{pmatrix} 42 & 5 & \cdots & 17 \\ 26 & 13 & \cdots & 1 \\ 18 & 21 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 10 & 37 & \cdots & 1 \end{pmatrix}$$

(b) Power curve P. Its rows correspond to the rows of S, while each columns represents one power domain.

$$\alpha = \begin{pmatrix} 1 & 0.5 & \cdots & 0.1 \\ 2 & 0 & \cdots & 1 \\ 1 & 0.5 & \cdots & 0 \\ 0.5 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 2 & \cdots & 0 \end{pmatrix}$$

(c) Power factor vector α. Its rows correspond to the columns of S, while its columns correspond to the columns of P.

Figure 3.1: Shape of the multidomain ILPM matrices.

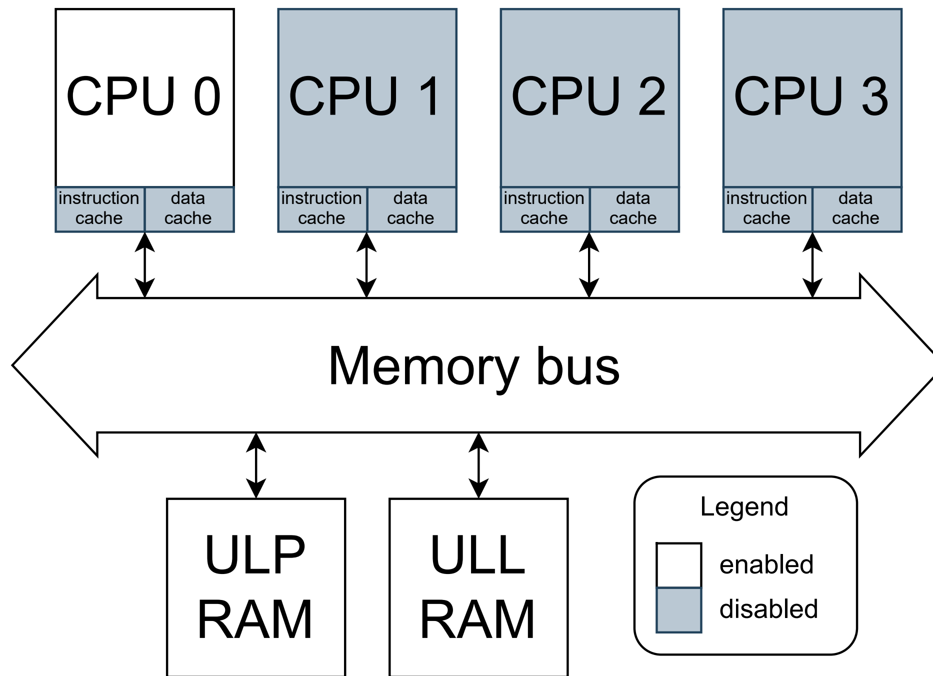


Figure 3.2: A diagram showing the rough layout of the TC4 compute domain.

Because of the disabled instruction cache, the active core spends a large part of the run time stalling while the next instruction is being fetched. This leads to some hard-to-predict instruction completion times. A wide range of test programs were simulated in the provided RTL testbench to create an instruction timing model, shown in table A.1 in appendix A. This timing model is not completely accurate, and is not made to predict the length of single instructions, but rather the execution time of a longer sequence of instructions.

3.1.1 Extracting a power curve

Extracting the power curve of a program is a two-step process: RTL simulation, and power analysis. After compiling or assembling the program for the TC4 platform, the RTL simulation is run using QuestaSim, exporting the switching activity in the form of a `.fsdb` file. This file is then analyzed using Synopsis PrimePower, and one power curve per module/domain under the compute domain is extracted in the `.frpt` format.

The `.frpt` files are converted to a comma separated values (CSV) format. Each row in the resulting files consists of two values, timestamp and power. There may be multiple rows in the power curve file for each row in a system trace, since PrimePower writes a new row in the `.frpt` file every time the power changes (up to a maximum of one row/ns). To correlate the power curve with a system trace, the timestamps of their rows need to match, so the power curve is normalized to fit the rows of the system trace.

For each row in the system trace, the normalization process reads the lines of the power curve which represent power usage within the time interval the row represents. The energy contributions from each line in the power curve is found, by multiplication of the power and the duration of each line, and accumulated. Care is taken when processing the first and last power curve line, as they might represent slices of time which are only partly within the system trace row's time interval. When every line has been processed, the total row energy

is divided by the time interval to get the row's power. The next row is then processed, and this continues until the power of all rows in the system trace has been found.

Because the instruction timing model is not completely accurate, the power curve and system trace can have different lengths, which could cause the feature occurrences to be correlated to the wrong parts of the power curve. Therefore, the normalization script also stretches the power curve to be the same length as the system trace in such a way that the power remains the same. The script also combines the different power curves into one table, and the result is one power curve file where each row shows the power usage of the corresponding row in the system trace, and each column shows the power curve of one domain.

3.2 TLM

The TLM model used is based on the open source RISC-V-TLM project[2], which is a model of a RISC-V core implementing the RV32IMACZifenceiZicsr ISA. It is written is SystemC, and uses the TLM library to model connections between the simulated hardware modules.

The original TLM model was modified in two main ways: the instruction timings found when analyzing TC4 were implemented as a delay after every instruction, according to table A.1; and the performance reporting system was overhauled to report a system trace with all desired ILPM features.

A time interval of $\Delta t = 10\,\mu\text{s}$ was chosen. For a program only executing instructions in the top row of table A.1, this means 20 cycles per row.

3.3 Features

A wide range of features were implemented in the TLM model, which fall into the categories listed in table 3.1. To examine the helpfulness of each type of feature, several models were created with different combinations of features.

The most basic model created was one with only single instruction features along with the time interval. All instructions in the RV32I ISA were modeled, except for FENCE, ECALL and EBREAK. This model was considered as a baseline for the other models, which all also use the single instruction and time interval features.

The next model created was *basic ilpm*, utilizing the time interval, single instruction and instruction swap features. The instruction swap features are all combinations of two types from table 3.2, with the ordering of the pair being important. As explained in section 2.2.1, this results in 12 instruction swap features. This model is along with the *single instructions* model considered a baseline for the other models.

One model was created for each of the feature types *branch*, *bitflips bus*, *bitflips regs* and *bitflips memory*. These models are a super set of *basic ilpm*, with their respective feature type added. The *branch* feature simply measures how many jumps occur. This is different from the number of jump and branch instructions, since branch instructions can be executed without jumping if the branch condition fails.

The *bitflips bus* features sum up the total hamming distance between consecutive values on three conceptual buses X, Y and Z. This is similar to the feature covered in section 2.2.3, with the difference that our model does not predict energy cycle by cycle but over a time interval. Since a goal of our model is to not be overly designed around one specific chip, these buses do not necessarily map one to one with buses in the hardware, but are solely based on the ISA, which is the reason behind the term "conceptual buses". Every time an

[2]Hosted at https://github.com/mariusmm/RISC-V-TLM

Table 3.1: Implemented ILPM feature types.

Type	Count
Time interval	1
Single instructions	37
Instruction swaps	12
Branch	1
Bitflips bus	3
Bitflips regs	1
Bitflips memory	1
Total	56

Table 3.2: The instruction types of different instructions as counted by the instruction swap features.

Type	Instructions
ALU	LUI, AUIPC, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND
LOAD	LB, LH, LW, LBU, LHU
STORE	SB, SH, SW
BRANCH	JAL, JALR, BNE, BEQ, BLT, BGE, BLTU, BGEU

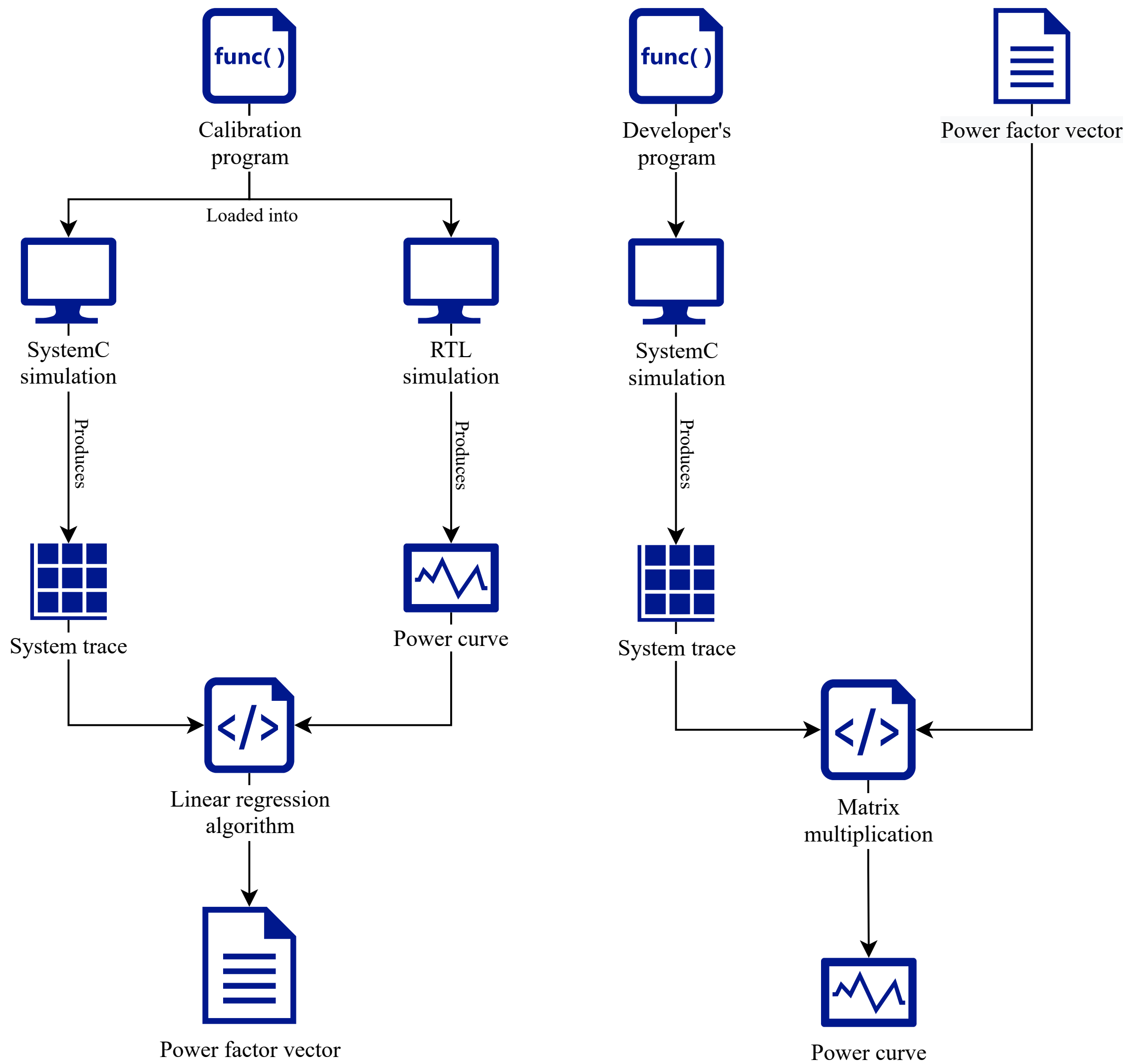
instruction is executed in our TLM model, it gets the data necessary for its execution from a combination of immediate values, CPU registers and memory. Depending on the encoding type of the instruction, these values are put into the X and Y buses, which represent the left and right operand. If the instruction has some sort of result, this value is put onto the Z bus. Every time a bus receives a new value, the hamming distance between the previous and new values is calculated and added to the total for the current time interval. A separate count is kept for each of the buses.

The *bitflips regs*, and *bitflips mem* features function the same way as the *bitflips bus* features, but it is the hamming distance of consecutive values in the CPU registers and memory registers respectively that is summed.

One final model, *all features*, was created with all the feature types utilized.

3.4 Calibration and prediction process

An overview of the process of calibration and prediction is shown in figure 3.3. A piece of calibration code is run both in the TLM and RTL simulation, which allows us to correlate the system trace and power curve using linear regression, resulting in a model, i.e. a PFV. For prediction, the model is applied on another program's system trace, obtained through a TLM simulation. There are slight differences from section 2.2 since our model is a multi-domain model.



(a) The calibration process. The same program is run in an RTL and a TLM simulation, and the resulting power curve and system trace are correlated to create the model.

(b) The prediction process. The model from the calibration process is used to predict the power curve of a new program.

Figure 3.3: The process of calibrating and utilizing the model.

3.4.1 Calibration process

As covered in section 2.2, the PFV α is extracted by minimizing the squared error in equation 2.2. This is the same as a linear regression model.

The equation is solved using a least squares method which allows for negative elements of α. The reason for not restricting α, given the argument that executing an instruction cannot create energy, is that there are features other than instructions in the system trace. Additionally, no row in the system trace contains only one feature, meaning that a negative total power will still be avoided, and if the power curve should ever become negative we know something is wrong with our model; the error does not stay hidden.

The model is able to predict the power usage of multiple domains by simply correlating the power curve of each domain to the system trace separately. Before the matrix equation is solved, the power curves are smoothed by convolution with a Hanning window, which has been normalized to preserve the average power. This is done to reduce the effect of noise, and the error from power spikes not being synchronized properly. The first and last few rows are also discarded to remove the effects of power-up and avoid having to write code specifically for the last row, which might only cover a fractional time interval.

3.4.2 Calibration code

The program used for calibration, known as the calibration code, is a program structured around running short loops designed to create many occurrences of single features. This allows the power contributions of each feature to be separated more easily. The loop bodies are made to take at least as long as the system trace time interval Δt to execute. It is written in assembly for a higher level of control of which features are present in each loop.

The program starts with loops of single instructions, then executes loops combining several kinds of instructions. This ensures that the single instruction and instruction swap features get enough data. Listing 3.1 gives an example of a loop of `add` instructions which use random registers not used for other purposes like loop control. Registers `x5` and `x6` are used as loop index variable and the loop limit respectively.

Listing 3.1: calibration_code.S loop structure

```
1  # ---- register loading ---- #
2  li x8, 1
3  li x9, 0
4  ...
5  li x14, 4
6  li x15, 2
7  # -------- main loop ------- #
8  addi x5, x0, 0   # set loop index
9  loop_ADD_0_0:
10     addi x5, x5, 1 # increment index
11     add x13, x11, x8
12     add x9,  x10, x11
13     ...
14     add x12, x15, x13
15     add x11, x8,  x12
16     bne x6, x5, loop_ADD_0_0 # x6 is set at program start
```

To ensure the *branch* feature's weight can be determined, the calibration code must make sure that some iterations of branch instruction loops end up branching and some not. This is achieved with the code in listing 3.2, where the index variable in `x5` is compared against a

constant in `x4` which is set once at the start of the program to be half of `x6`. This ensures that the branch condition is true for at least one iteration of every branch loop, and false for at least on iteration. `nop` instructions have been inserted as padding since there are strange timing effects that occur when jumping several instructions in a row (these effects were not thoroughly enough researched to put in table A.1).

Listing 3.2: calibration_code.S loop structure

```
1  addi x5, x0, 0   # set loop index
2  loop_BLT_0_0:
3      addi x5, x5, 1 # increment index
4      blt x5, x4, label_040
5      nop
6      nop
7  label_040:
8      nop
9      nop
10     blt x5, x4, label_041
11     nop
12     nop
13     nop
14 label_041:
15     ...
16 label_058:
17     nop
18     nop
19     nop
20     blt x5, x4, label_059
21     nop
22     nop
23     nop
24 label_059:
25     bne x6, x5, loop_BLT_0_0
```

The calibration code must also provide a wide range of bus, register and memory activity, so the model is able to calibrate the weights of the *bitflips* feature types properly. Each instruction has several consecutive loops associated with it, where the first loop attempts to facilitate a low activity and the last a high activity. The first method to achieve this is varying the values in the utilized registers; before every loop, each register used in the loop (except `x0`, which is special) is overwritten, as shown in line 2-6 in listing 3.1. For the first loop, the written values are all small numbers represented by only a few bits. For the last loop, the entire range of the 32-bit register is used. The second method is restricting the immediate values in the looped instructions in the same way: the first loop uses only smaller values, while the following loops use progressively larger values.

The different types of *bitflips* features will naturally have different numbers of occurrences for the different instructions, so no actions are taken to make their power contributions separable.

After the single-instruction loops, there is a section of multi-instruction loops, at least one loop per *instruction swap* feature. A loop cannot contain occurrences of only one instruction swap, so different sets of two and three instructions are picked in such a way that all features occur at least once and all the features' power contributions can be separated from each other. Since branches and bitflips are covered well by the single instruction loops, there is only one loop per set of instructions.

3.4.3 Prediction

Prediction requires a system trace and a previously calibrated model, i.e. a PFV. The prediction is done using a simple matrix multiplication as shown in equation 2.3.

Chapter 4

Evaluation

The main evaluation criterion for the model is its accuracy measured in relative average error $\bar{\epsilon_r}$, which is defined by equation 4.1.

$$\begin{aligned} \bar{\epsilon_r} &= \frac{1}{N\bar{P}} \sum_{i=0}^{N} \epsilon_i = \frac{1}{N} \sum_{i=0}^{N} (\hat{P}_i - P_i)/\bar{P} \\ &= \frac{1}{N} \sum_{i=0}^{N} \hat{P}_i/\bar{P} - 1 \end{aligned} \tag{4.1}$$

where N is the number of rows in the power curve, and P_i and $\hat{P}_i$ are the real power and estimated power respectively of row i. Other stats like root-mean-Square (RMS) error and R^2 are also calculated per model, domain and benchmark. These stats might provide some insight into the model, but they are not as useful as relative average error as evaluation criteria, since the main use case of our model is predicting total energy usage; the ups and downs of the power curve are less important.

The stats for all domains are calculated separately. This allows us to get a better picture of the model's strengths and weaknesses, i.e. whether there is one domain the model fails to predict power usage accurately for.

A separate R^2 value is calculated which compares the models' ability to predict the average power usage of a benchmark on each domain. The R^2 value shows a model's predictive power in comparison with the constant model which would best fit the data. A value of 1 means the model predicted the data perfectly, while 0 means the model was only as good as a constant guess. A negative value means the model was worse at predicting the data than a constant model. Equation 4.2 shows the definition of R^2 for a model f predicting labels y from features x.

$$R^2 = 1 - \frac{\sum_i (f(x_i) - y_i)^2}{\sum (\bar{y} - y_i)^2} \tag{4.2}$$

4.1 Benchmarks

Six benchmarks, listed in table 4.1, were used to evaluate the accuracy of our model. The code for all the benchmarks, except the Coremark test, is listed in appendix B. The same benchmarks are all adapted from [Wan17].

Table 4.1: Benchmarks used to evaluate the model.

Benchmark	ref.	Description
Coremark	[EEM22]	An embedded power benchmark inspired by dhrystone, developed to test power usage in low power micro-controllers and CPU cores.
Fib	Appendix B.1	Calculates the 15th Fibonacci numbed recursively.
Fir	Appendix B.2	Apply a finite impulse response filter to an array.
Hanoi	Appendix B.3	Simulate a game of the towers of Hanoi.
Quicksort	Appendix B.4	Sort an array using the quicksort algorithm.
Tak	Appendix B.5	A recursive-based benchmark function.

4.1.1 Data dependency

Another test was performed to investigate how well the model can fit a heavily data dependent sequence of `addi` instructions. The test consists of four parts with 256 `addi` instructions; the difference between the parts is only the operands. The first part is `nop` instructions, implemented in RISC-V as `addi x0, x0, 0`. These should cause almost no switching activity. The second part is `addi x11, x11, 1`, which should cause a small amount of activity. The third part starts with loading a 1 into `x11`, and then alternates between decrementing twice and incrementing twice. This will cause a lot of transitions between -1 (`ffffffff`) and 0 (`00000000`), which is a high level of switching activity. The final part consists of `addi x11, x11, <X>`, where `<X>` is a number between 1000 and 2023. This should cause a medium level of activity.

Chapter 5

Results

5.1 Power factor vector

The resulting power factor vector from calibration the *all features* model is given in figure 5.3. The weights have been normalized in such a way that the energy cost of keeping the chip powered on for 10 ns is 1. Figures 5.1 and 5.2 show the same information for the *basic ilpm* and *branch taken* models. In order to be visible, the listed weight of the *bitflips* features are per 10 bit flips.

5.2 Individual statistics

Tables showing the relative average, maximum and RMS errors of the power estimation using the different models can be found in appendix C. Each table shows the statistics for each domain and model for one benchmark. The subscript r denotes that an error is relative to the average of the real (RTL) power. Table 5.1 shows the real (RMS) average power used by each domain during simulation of each benchmark. The values have been normalized within each domain, such that the average power of the calibration code is 1.

Table 5.2 shows how well the instruction timing table predicted the execution time of the different benchmarks. Figure 5.4 shows what a power curve of the calibration code looks like, alongside a prediction by the *single instructions* model. fig. 5.5 shows the power curves for the other test benches and domains compared to the prediction of the *all features* model. To make the plots more readable, all the curves have been smoothed using a Hanning window. The radius of this window was 100 for the *coremark* testbench, and 10 for the rest.

5.3 Data dependency test

Figure 5.6 shows real and predicted power for each domain when running the data dependency test. The power has been normalized.

5.4 Model comparisons

Figure 5.7 shows the relative error vs real (RTL) average power usage of the different tests as scatter plots, with each point being one model predicting one benchmark. Each plot covers one domain. The real power has been normalized as in table 5.1. The R^2 for the average power usage prediction of each model in each domain is shown in table 5.3. This R^2 differs from the R^2 in the tables in appendix C, which show how well a model's prediction fits the

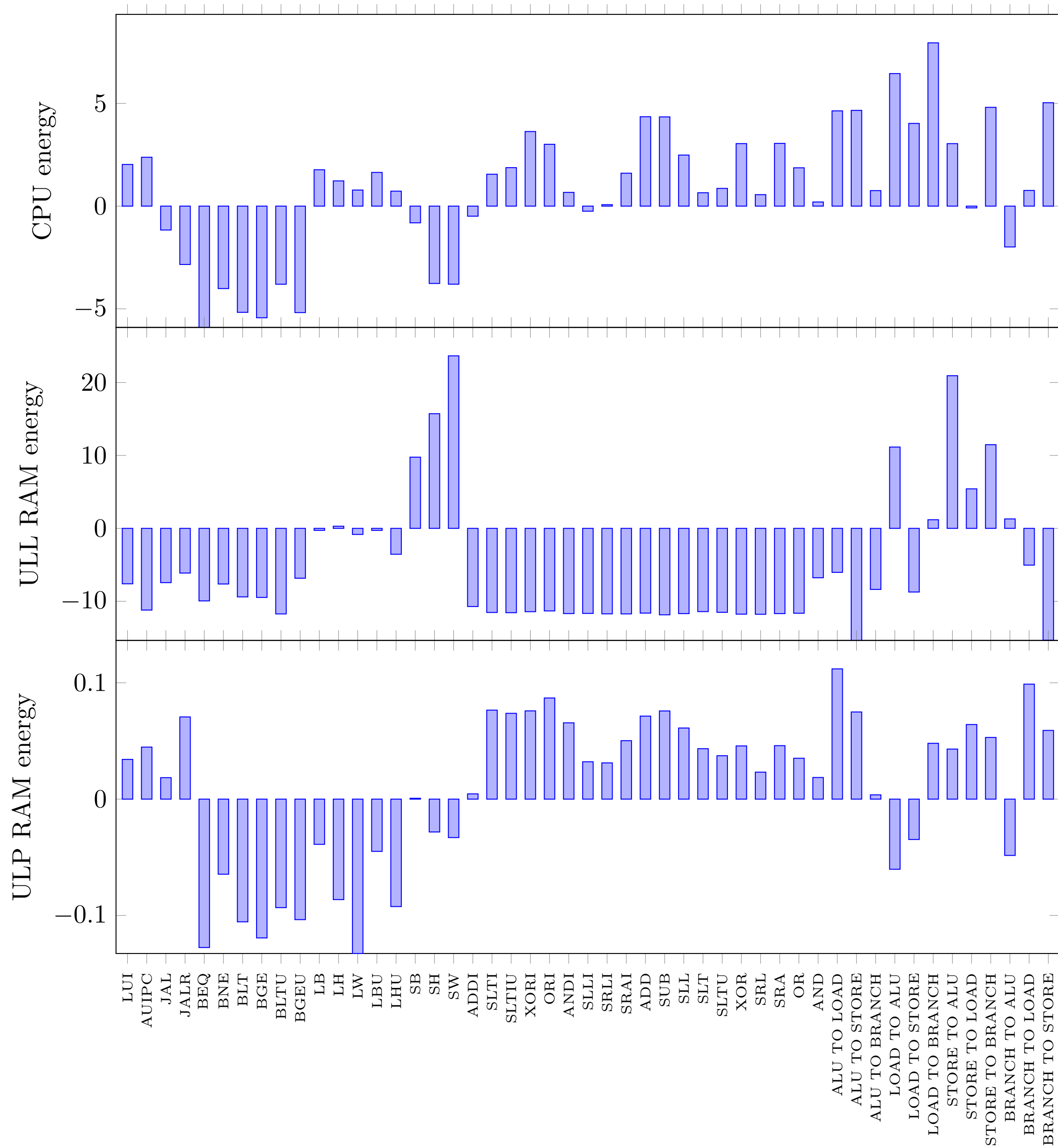


Figure 5.1: The power factor vector for the CPU module of the *basic ilpm* model. Normalized to the energy cost of delta time (10 ns).

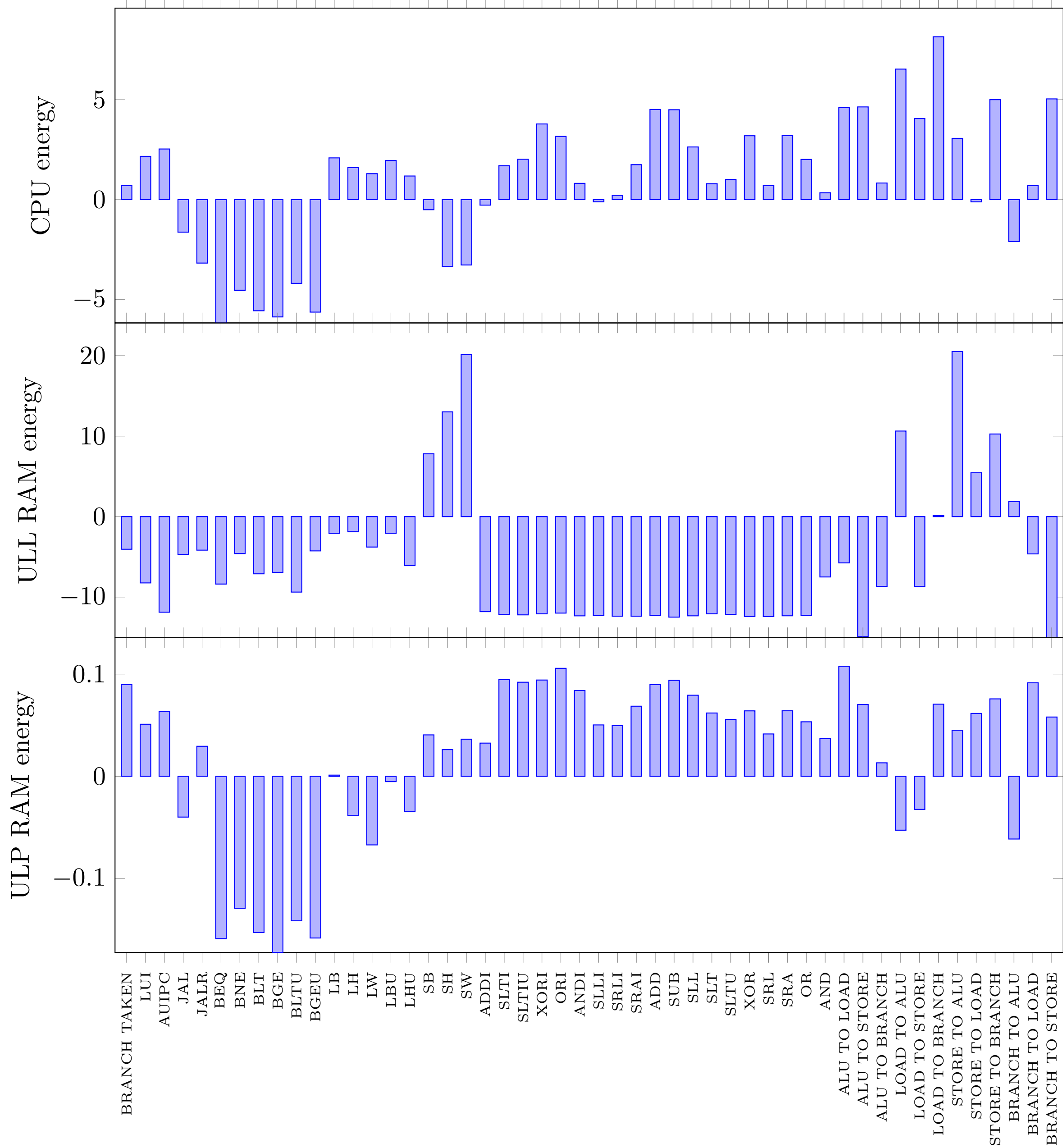


Figure 5.2: The power factor vector for the CPU module of the *branch taken* model. Normalized to the energy cost of delta time (10 ns).

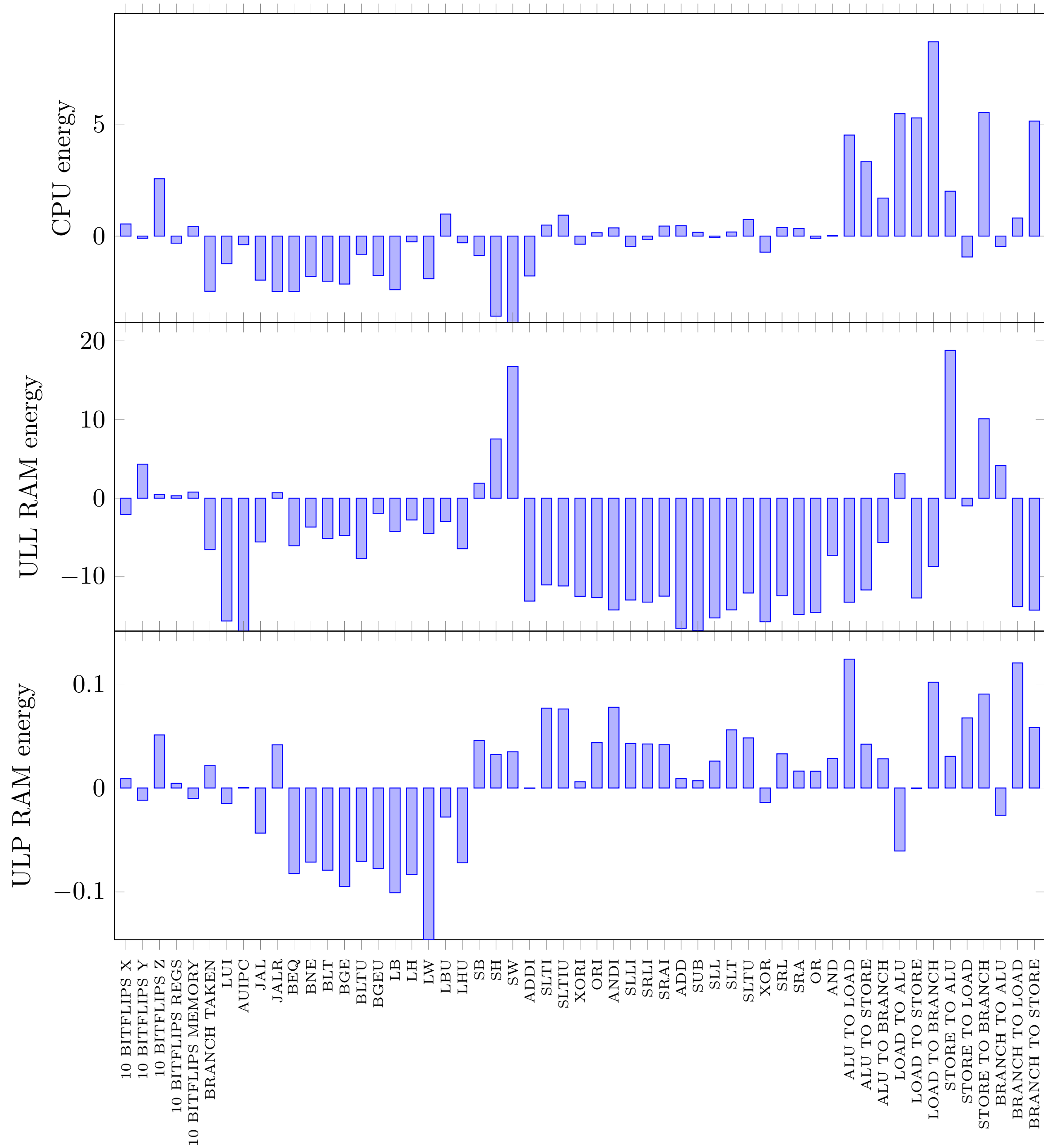


Figure 5.3: The power factor vector for the CPU module of the *all features* model. Normalized to the energy cost of delta time (10 ns).

Table 5.1: Average power by domain for different all benchmarks extracted from RTL simulation. Normalized within each domain to the average power of the calibration code.

benchmark	CPU	ULL RAM	ULP RAM
calibration code	1.000	1.000	1.000
coremark	1.047	1.009	1.005
fib	1.028	1.048	1.002
fir	0.988	0.981	0.970
hanoi	0.916	0.972	0.902
quicksort	0.976	0.978	0.944
tak	0.993	1.064	0.975

benchmark	$T_{\mathrm{TLM}}/T_{\mathrm{RTL}} - 1$ (%)
calibration code	0.27
coremark	2.26
fib	-0.50
fir	-1.21
hanoi	-2.04
quicksort	-3.99
tak	-2.32

Table 5.2: The accuracy of the instructing timing for different benchmarks. Found by dividing the execution time predicted by the model by the real execution time from the RTL simulation.

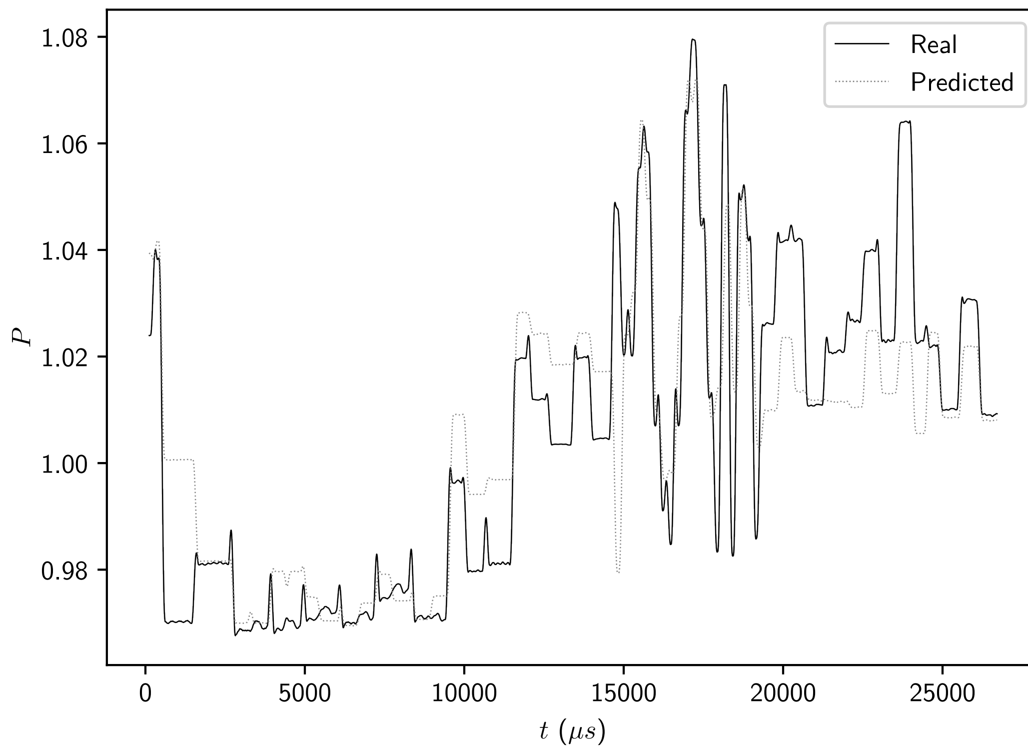


Figure 5.4: The real vs predicted power used by the CPU when running the calibration code. The model used for prediction is *single instructions*. The power has bee normalized to the average of the real power usage.

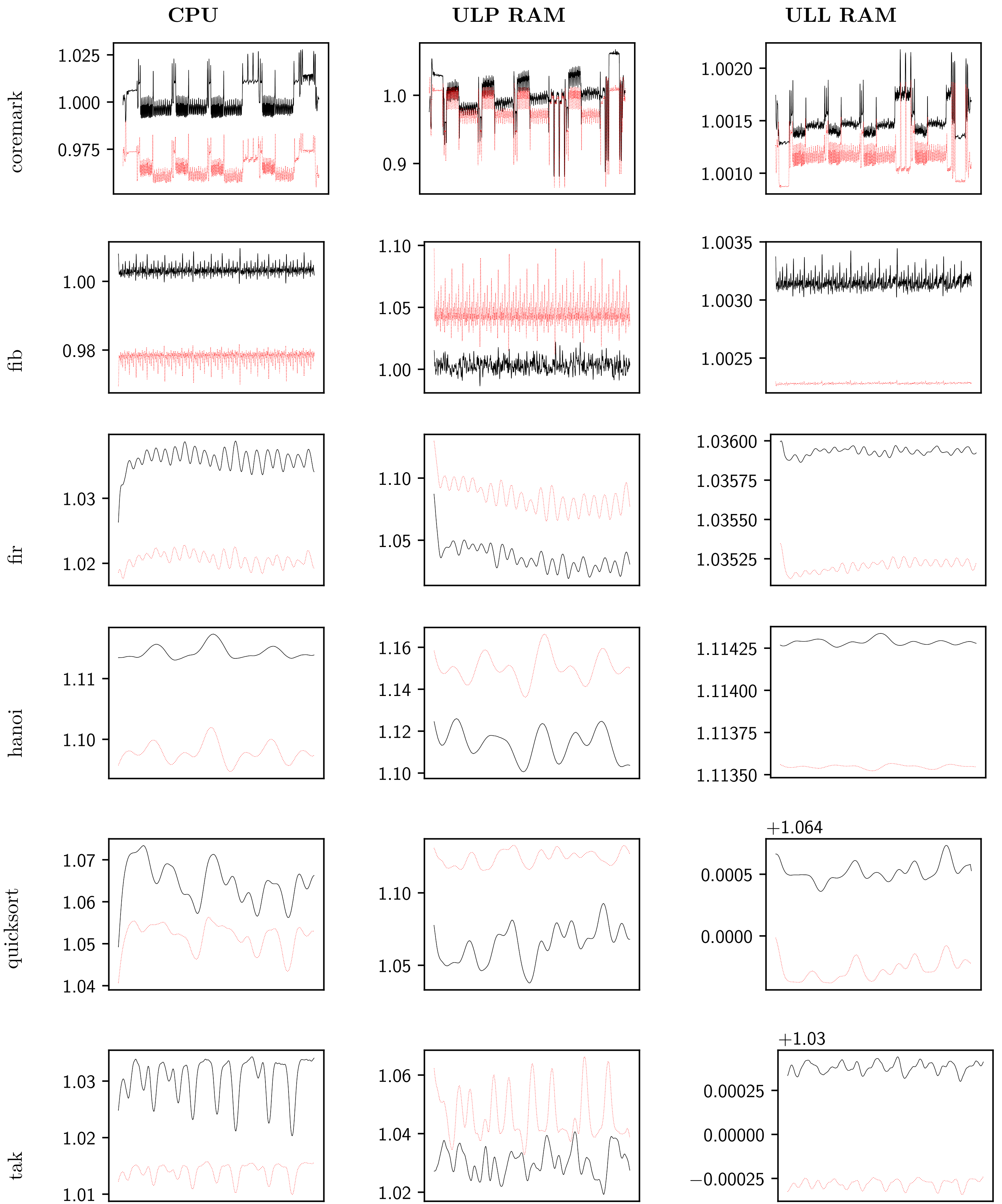
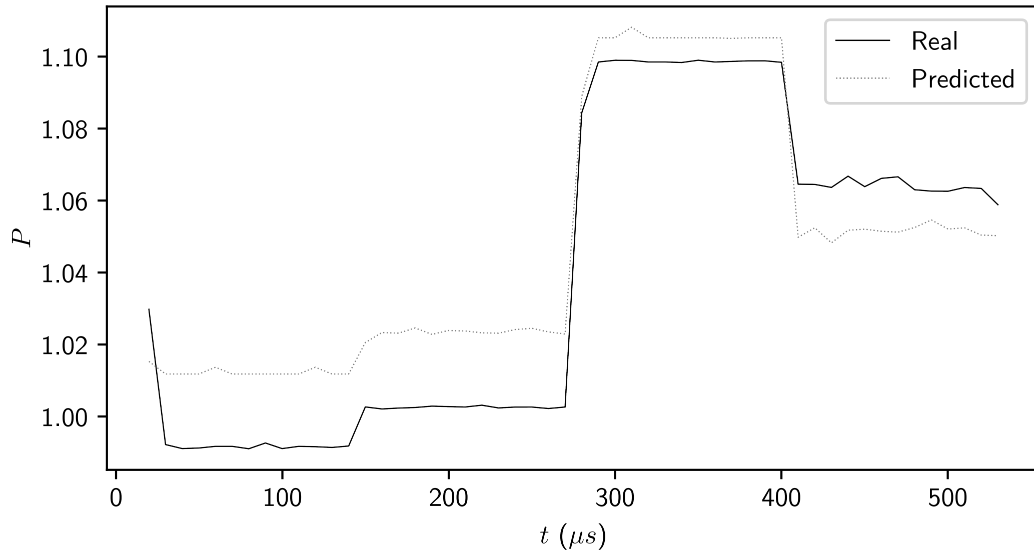
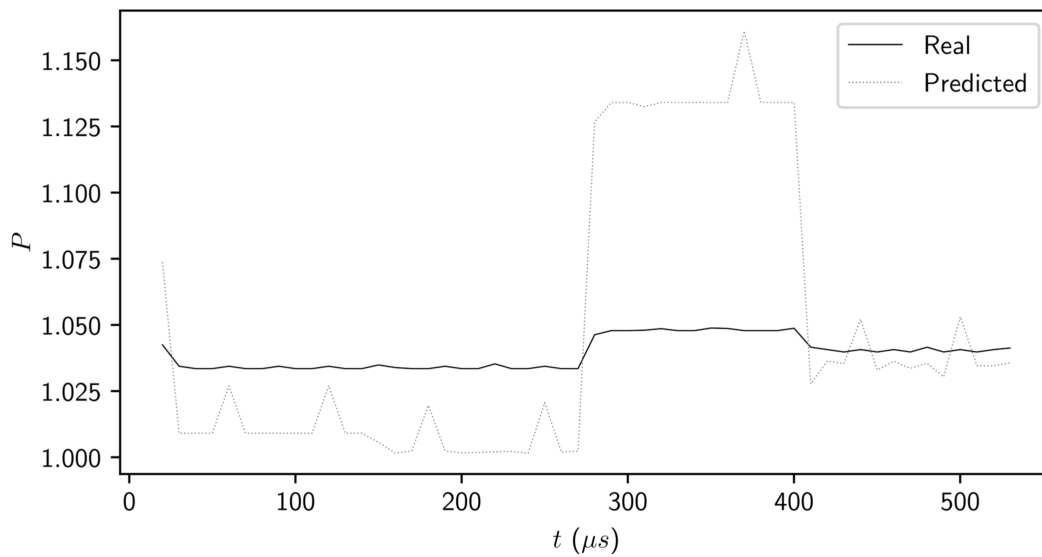


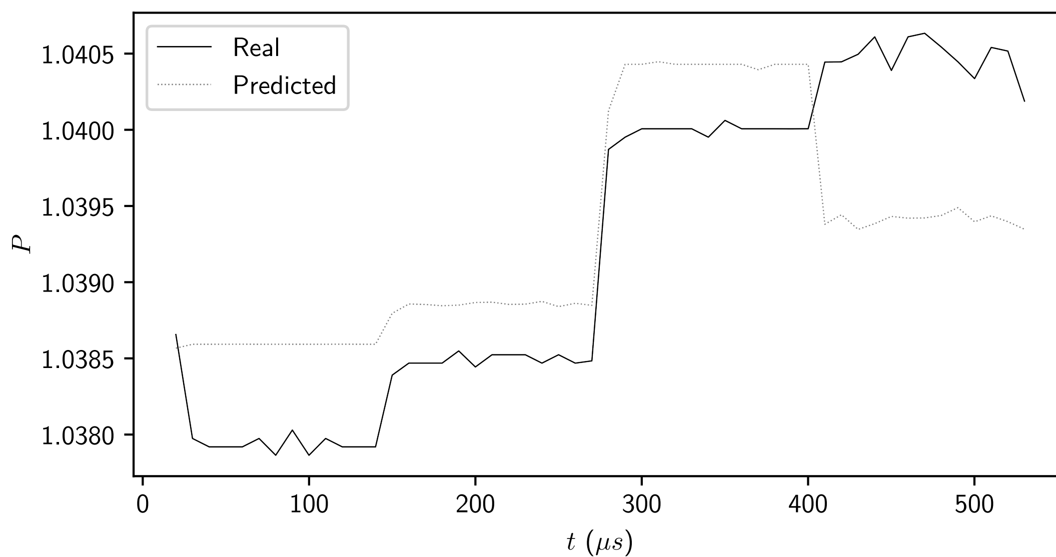
Figure 5.5: Real (black) vs predicted (red, dotted) power used by the domains when running different benchmarks. y-axis shows power, x-axis shows time. The *all features* model was used for all predictions.



(a) CPU



(b) ULL RAM



(c) ULP RAM

Figure 5.6: Real vs predicted power used by the domains when running the data dependency test. The *all features* model was used for predictions.

Table 5.3: R^2 for all model across all domains. This R^2 is based on the ability of the models to predict average power over a program's execution.

model	ULL RAM	ULP RAM	CPU
only single instr	-0.2810	0.9997	0.6359
basic ilpm	-0.5198	0.9997	0.6896
branch taken	-0.1129	0.9998	0.6979
bitflips bus	-0.9085	0.9996	0.7749
bitflips regs	-0.4325	0.9997	0.7388
bitflips memory	-0.5088	0.9997	0.6875
all features	-0.1456	0.9997	0.7298

variation over time within a single benchmark, because this R^2 shows how well the model beats an constant cost model when predicting average power usage.

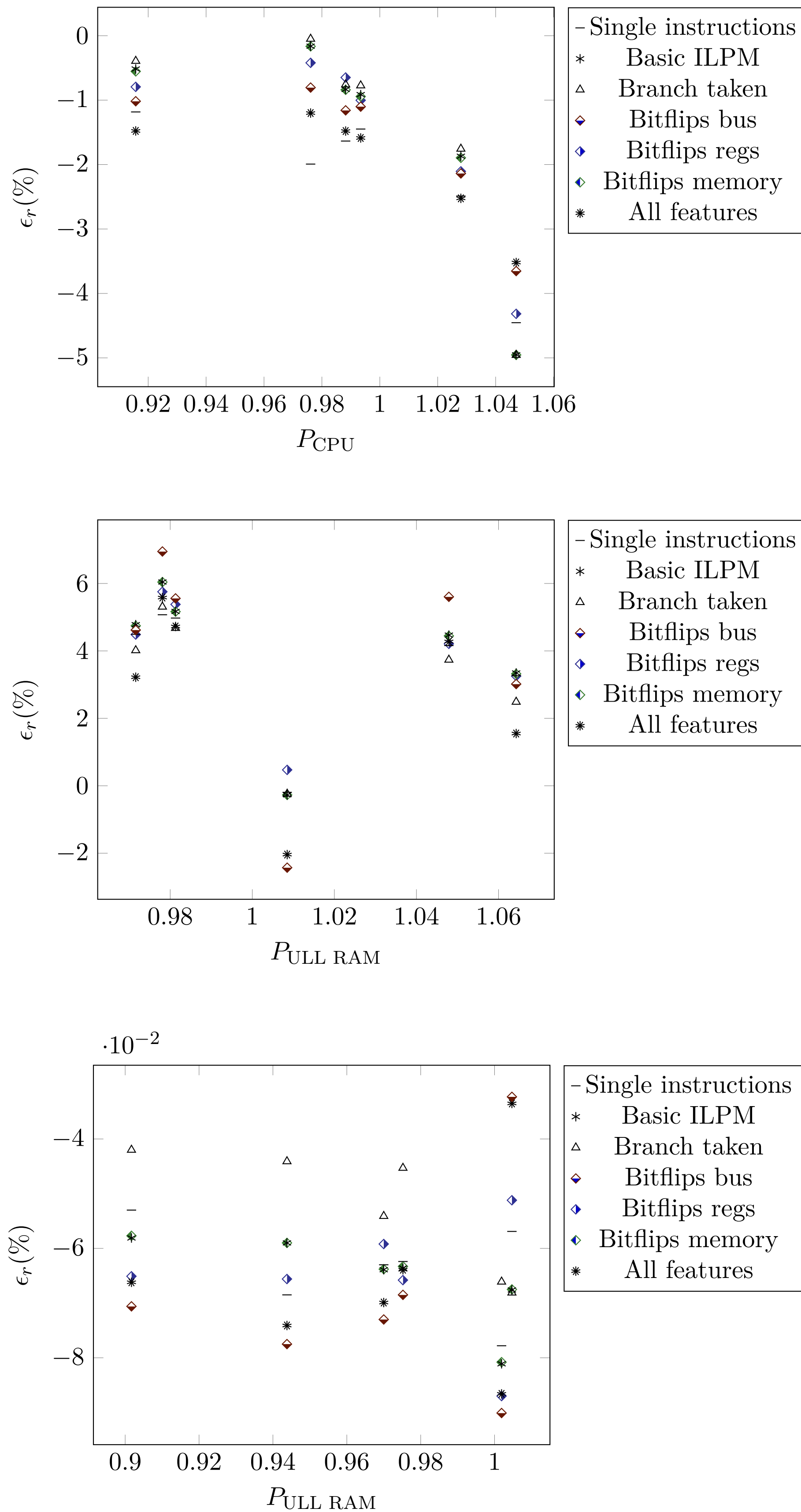


Figure 5.7: Relative error by actual power in the different domains. The actual power is normalized to the calibration code average power.

Chapter 6

Discussion

6.1 Interpretation of the results

6.1.1 Differences between the power domains

Table 5.3 gives a good overview over how well the different models predict the average power usage of the different benchmarks. We see that all the models predicted the power usage of the ULP RAM domain a lot better than an constant power model, but that no model was better than a constant power model when predicting power usage in ULL RAM. Negative R^2 values suggest that the power usage of the calibration code was unrepresentative of a typical workload. Specifically, the calibration code might not have enough cases of jumps to function starts followed by a large number of stores to different places in memory, which is typical for un-optimized function calls.

Tables 5.1 to 5.3 show the difference in energy costs of instructions between the domains. As expected, the domains have different features which have high and low energy costs; the low-leakage memory uses little power outside load/store unit (LSU) instructions, while the CPU has higher energy costs for ALU instructions. The low-power memory's power usage is dominated by the baseline power from just being powered up.

When predicting CPU power, all models were better than constant, but not as good as in ULP RAM. The two least feature-rich models (*only single instr* and *basic ilpm*) were also the worst performing in the CPU domain. However, the *all features* model was not the best performing, scoring lower than *bitflips regs* and *bitflips bus*. This suggests that the models were over-fitting to the calibration code, wrongly assigning energy costs to features that just happened to overlap with a spike or drop in energy.

In conclusion, not much work is needed on modeling the ULP RAM, as it has the lowest relative error and the highest R^2 values. The CPU modeling was largely successful, but the model is prone to over-fitting. Finally the ULL RAM model consistently does worse than a constant power model, which is a sign of non-representative calibration code.

6.1.2 Adding features and over-fitting

The PFVs in figs. 5.1 and 5.3 show the above-mentioned over-fitting. The *basic ilpm* model assigns very similar energy costs to the ALU instructions from `ADDI` to `OR` in the ULL RAM domain. This corresponds with reality, as the memory module's energy usage should not be dependent on the operations internal to the CPU. In the *all features* model, however, the energy cost of these instructions vary quite a bit in the ULL RAM. This shows that the addition of more features can lead to a model less representative of reality. Ideally, the calibration code covers so many different cases that the calibration process can separate

the signal from the noise and end up with values that make sense and therefore have more predictive power.

We see the opposite effect between figure 5.1 and 5.2, where the addition of *branch taken* makes the energy costs of the instructions between `BEQ` and `BGEU` more similar for the ULP RAM domain.

Some features in *all features* are also suspiciously negative. While it make sense for some features like single instructions and instruction swaps to have negative weights, it does not make sense for bit flip features to be negative. This is because a higher number of bit flips should mean a higher switching activity and a higher power usage. The cause of the negative weights might be that a high number of bit flips on one bus is correlated with a high number of bit flips on other buses in the calibration code.

In conclusion, an increase in the number of different features must be met with appropriate additions and changes to the calibration code.

6.1.3 Data dependency

Figure 5.6 gives us an idea of the data-dependent nature of the power usage in the different domains. All the domains are data dependent in some form, even when all the executed instructions are `addi`. The power usage of the CPU is as expected, going from almost no activity to low to high to medium. The data dependency of the low leakage memory is overestimated by the model, and the power curve has largely the same shape as the CPU's power curve. The overestimation is a sign that the calibration code pairs high bus activity with memory accesses, and that the calibration process does not have enough data to separate them. The domain still exhibits some data dependency, which could mean that some part of the memory module is exposed to the activity of the CPU buses, even when the instructions executed do not use the memory.

The low-power memory's power curve is extra interesting, as the medium activity case has a higher power usage than the high activity case. There could be some power contribution from the hamming distance between consecutive instructions, but this cannot be the only data dependent contribution, as the no activity and low activity regions of the power curve have the same hamming distance (zero) between instructions.

More research could be done into this topic, but the accuracy gain from doing so is minimal. The largest error (ULL RAM 300-400µs) will most likely be fixed with better calibration code.

6.1.4 Calibration code

So far, we have seen that improving the calibration code could increase the models' accuracies. Figure 5.5 supports this by showing that the model consistently either under- or overestimates the power usage within a single domain. For example, adding 1 % to the prediction of the CPU's power would increase the accuracy of all the tests. This means that if the calibration code had a higher average power usage in the CPU (which is more representative of the benchmarks), the model would have a higher accuracy. It is also possible that there is something about the benchmarks that make their power curves unrepresentative for programs in general. This seems less likely than the calibration code being bad, as the benchmarks are varied in their workload, are compiled from a higher level language (which most code will be) and the model consistently either over- or underestimates the power within each domain. The only exception is the power usage of ULP RAM when running coremark, which is underestimated, while the power usage of the rest of the benchmarks are overestimated.

6.1.5 Accuracy

From the tables in appendix C we see that the model with the best worst-case performance for the CPU domain was *all features* with a worst case error of $-3.52\,\%$ running coremark. For the ULL RAM the best worst-case was *single instructions* with $5.07\,\%$ error running quicksort. The model achieving the best worst-case for the ULP RAM was *branch taken* with $0.07\,\%$ error running coremark.

However, worst case accuracy is not the only measure of a model's accuracy; the R^2 value in table 5.3 is preferred.

6.2 Weaknesses

This study has some weaknesses that should be pointed out. Firstly, no cache was utilized in any benchmark or calibration. This means there was no analysis of the effect of cache hits and misses on power usage. The CPU used is also relatively simple, with only a two-stage pipeline.

There was also no analysis of total energy used, only average power was used. This makes a difference in accuracy when considering the accuracy of the instruction timing model shown in table 5.2.

The paper also only used six benchmarks. More benchmarks means more confidence that the worst-case accuracy and R^2 statistic is accurate.

The model was calibrated to, and tested against, the power reported by an RTL simulation, which itself might not be accurate to the real power consumption. This was justified by the fact that it is the ILPM model's ability to reflect a program's power curve that is being tested, and that the actual accuracy of the 'real' power curve is irrelevant. However, it could be the case that a simulation of a fully floor-planned and routed chip yields power curves that are not only more accurate to the real world, but also harder to predict because of some unforeseen effects.

It is also a weakness that the model only covers one CPU core and two memory modules. Ideally, the model should have covered the entire chip, but this would require building a more advanced TLM model of the chip, extending the power model to include features for peripherals, and writing calibration code which can give the calibration process the data needed to separate the power contributions from the different features.

6.3 Future work

A lot of work can be done to further analyse and improve upon the model.

6.3.1 Further analysis

There has been no analysis of how well different calibration methods work. It is possible to look into the possibility of improving the calibration code, or the effects of removing parts of the calibration code. More experimentation with different linear regression methods can be done; this model used a normal least-squares solver, but it is possible to use a non-negative solver, or a solver which prefers a lower number of non-zero weights.

6.3.2 Model improvements

One feature that can be added to the model is hamming distance of consecutive instructions, and hamming distance between consecutive values in the PC register. Both of these will affect the power used by the memory buses, as they are the data and address of the instruction memory respectively.

More thought can be put into restricting some features to some power domains, to mitigate the tendency to over-fit to the calibration code. This will also reduce the size of the model, which may be useful if it starts covering multiple cores and peripherals.

Much work can be done to improve the calibration process. A (relatively) simple first step is to allow the calibration code to be multiple programs, and fit the model to all their power curves. This removes the need for loop bodies of only 20 instructions, since this was limited by program memory size and the number of different kinds of loops. With several programs, one program can be only `addi`, and one only `lb`, etc.

The instruction timing model can be improved. Either more work can be done to make it accurate, which requires a lot of repeated manual work and a complete restart if the target CPU is updated or changes. Another idea is to calibrate the timing model like we calibrate the power model, creating an Instruction Level Timing Model of sorts. This would automate a large part of the work when switching target CPUs. Keeping the system trace and power curve synchronized while calibrating the model could be done by extracting the waveform of the program counter from the RTL simulation, and using it to synchronize with the TLM simulation.

The largest improvement in terms of practical usefulness is to model an entire chip. This means extending the TLM model to include several cores in parallel, writing a cache simulator for the particular chip, modeling the interface to peripherals, looking at simulating sleep modes and much more. For a simpler micro-controller, it may be enough to keep the ILPM for a single core, but ultra low power devices have a plethora of methods for energy saving (sleep modes, DVFS, specialized peripherals, etc.), and these must be considered when making a useful power model.

6.4 Conclusion

In this paper, we showed that the power usage of a CPU, an ultra-low-power memory and an ultra-low-leakage memory can be modeled with ILPM with a maximum error of 3.52 %, 5.07 % and 0.07 % respectively. An R^2 value of 0.775 was reached in terms of predicting the average power usage in the CPU while running six different benchmarks.

We found that adding more features to the model can make the model more accurate, but any additional features must be met with more code to calibrate the weights of these features, to avoid over-fitting. All features added to the model alone increased its R^2 value, but the model's predictive power declined when too many features were added.

Additionally, care should be taken that the code used to calibrate the model has instruction patterns which reflect the instructions patterns of real programs, lest the model systematically under- or overestimates the power usage.

Appendix A

Instruction timings

Table A.1: Instruction completion times for all modeled instructions. A 0-aligned address is one which ends in 0 or 8. 4-aligned addresses end in 4 or c. One cycle is 62.5 ns.

Instructions	Timing
LUI, AUIPC, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND	8 cycles

JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU	The timing depends on whether the instruction ends up jumping, and on the alignment of the PC and destination address.
	if no jump **then**
	  8 cycles
	else
	  if PC 0-aligned **then**
	    if destination 0-aligned **then**
	      16 cycles
	    else
	      20 cycles
	    end if
	  else
	    if destination 0-aligned **then**
	      20 cycles
	    else
	      24 cycles
	    end if
	  end if
	end if

<table>
<tr>
<td>LB, LBU, SB</td>
<td>LSU instructions affect the timing of consecutive LSU instructions. This is modeled by a counter c which increments by one on any 8-bit LSU instruction, by two on any larger LSU instruction, and decrements on any non-LSU instruction. c is always a non-negative integer no greater than 2. The timing is also affected by the alignment of the program counter.

if $c = 0$ **then**
    20 cycles
else
    if PC 0-aligned **then**
        16 cycles
    else
        20 cycles
    end if
end if</td>
</tr>
<tr>
<td>LH, LHU, SH</td>
<td>LSU instructions which operate on data longer than one byte have timings which depends on the alignment of the data address a.

if $a\%2 = 0$ **then**
    like an 8-bit LSU instruction
else
    if $c < 2$ **then**
        28 cycles
    else
        if PC 0-aligned **then**
            28 cycles
        else
            32 cycles
        end if
    end if
end if</td>
</tr>
</table>

LW, SW	**if** $a\%4 = 0$ **then**
	    like an 8-bit LSU instruction
	else if $a\%4 = 2$ **then**
	    like a 16-bit LSU instruction
	else
	    if $c < 2$ **then**
	        40 cycles
	    else
	        if PC 0-aligned **then**
	            40 cycles
	        else
	            44 cycles
	        end if
	    end if
	end if

FENCE, ECALL, EBREAK	Not analyzed. Just uses the standard 8 cycles.

Appendix B

Benchmarks

This appendix contains the source code for five of the benchmarks that the model was evaluated with. All are versions of the benchmark code in appendix A.1.4 of [Wan17].

Listing B.1: fir.c

```
1  // fib C code
2  #include "tc.h"
3
4  #pragma GCC optimize ("-O0")
5  int fib (int n) {
6      if (n < 2)
7          return n;
8      else
9          return (fib(n-1) + fib(n-2));
10 }
11
12 int main() {
13     int a;
14     a = fib(15);
15     return 0;
16 }
```

Listing B.2: fir.c

```
1  // fir C code
2  #include "tc.h"
3
4  #define F_LENGTH 20
5  #define K_LENGTH 5
6  #define K_RADIUS (K_LENGTH-1)/2
7  #define I_LENGTH F_LENGTH + K_RADIUS * 2
8  #pragma GCC optimize ("-O0")
9  void firFixed( int *coeffs, int *input, int *output, int length, int filterLength ) {
10     int acc; // accumulator for MACs
11     int *coeffp; // pointer to coefficients
12     int *inputp; // pointer to input samples
13     int n;
14     int k;
15     // apply the filter to each input sample
16     for ( n = 0; n < length; n++ ) {
17         // calculate output n
18         coeffp = coeffs;
```

```
19          inputp = &input[n];
20          acc = 0;
21          // perform the multiply-accumulate
22          for ( k = 0; k < filterLength; k++ ) {
23              acc += (*coeffp++) * (*inputp--);
24          }
25          output[n] = acc;
26      }
27  }
28
29  int main(void) {
30      int input  [I_LENGTH] = {
31          0, 0, 0, 0, 1,
32          1, 1, 1, 1, 5,
33          1, 1, 2, 3, 4,
34          2, 1, 4, 2, 1,
35          5, 2, 0, 0
36      };
37      int output [F_LENGTH];
38      int coeffs [K_LENGTH] = {0, 100, 500, 100, 200};
39      firFixed(coeffs, input + K_RADIUS*2, output, F_LENGTH, K_LENGTH);
40      return 0; // 0 means test success
41  }
```

Listing B.3: hanoi.c

```
1   // hanoi C code
2   #include "tc.h"
3
4   #pragma GCC optimize ("-O0")
5   int Hanoi ( int from, int to, int use, int howmany ) {
6       if (howmany == 1) {
7           return 1;
8       } else {
9           int imovs =0;
10          imovs += Hanoi(from, use, to, howmany -1);
11          imovs += Hanoi(from, to , use, 1);
12          imovs += 1;
13          imovs += Hanoi(use, to, from, howmany-1);
14          return imovs;
15      }
16  }
17
18  int main (void) {
19      Hanoi(3,2,1,5);
20      return 0;
21  }
```

Listing B.4: quicksort.c

```
1   // quicksort C code
2   #include "tc.h"
3
4   #pragma GCC optimize ("-O0")
5   void quicksort(int list[], int m, int n) {
6       int key,i,j,k,temp;
7       if(m < n) {
8           k = (m+n)/2;
```

```
9           // swap [m] and [n]
10          temp = list[m];
11          list[m] = list[k];
12          list[k] = temp;
13          key = list[m];
14          i = m + 1;
15          j = n;
16          while(i ≤ j) {
17              while((i ≤ n) && (list[i] ≤ key)) i++;
18              while((j ≥ m) && (list[j] >  key)) j--;
19              if(i < j) {
20                  // swap [i] and [j]
21                  temp=list[i];
22                  list[i]=list[j];
23                  list[j]=temp;
24              }
25          }
26          // swap [m] and [n]
27          temp=list[m];
28          list[m]=list[j];
29          list[j]=temp;
30          // recursively sort the lesser list
31          quicksort(list, m,    j-1);
32          quicksort(list, j+1, n   );
33      }
34  }
35
36  int main() {
37      const int MAX_ELEMENTS = 15;
38      int i = 0;
39      int list[25] = {
40          19, 74, 17, 33, 94,
41          18, 46, 83, 65,  2,
42          32, 53, 28, 85, 99,
43          11, 68, 67, 29, 82,
44          21, 62, 90, 59, 63
45      };
46      // sort the list using quicksort
47      quicksort(list, 0, MAX_ELEMENTS-1);
48      return 0;
49  }
```

Listing B.5: tak.c

```
1   // tak C code
2   #include "tc.h"
3
4   #pragma GCC optimize ("-O0")
5   int tak( int x, int y, int z ) {
6       int a1, a2, a3;
7       if (x ≤ y) return z;
8       a1 = tak(x-1,y,z);
9       a2 = tak(y-1,z,x);
10      a3 = tak(z-1,x,y);
11      return tak(a1,a2,a3);
12  }
13
14  int main(void) {
15      tak(10,5,3);
```

```
16      return 0;
17  }
```

Appendix C

Statistics

This appendix contains large tables of results from running benchmarks on the different models. The value for R^2 was rounded to $-\inf$ if the calculated value surpassed -3000.

Table C.1: Statistics for the calibration code benchmark.

domain	model	$\bar{\epsilon_r}$ (%)	$\max\{\lvert\epsilon_r\rvert\}$ (%)	RMS(ϵ_r) (%)	R^2
ULL RAM	**bitflips regs**	0.0007	13.3917	1.9465	0.9756
	bitflips memory	0.0007	13.4222	1.9468	0.9756
	all features	0.0007	13.2055	1.8453	0.9780
	only single instr	0.0002	13.7805	2.0224	0.9736
	basic ilpm	0.0007	13.4366	1.9497	0.9755
	branch taken	0.0007	13.4318	1.9353	0.9758
	bitflips bus	0.0007	13.2510	1.8697	0.9775
ULP RAM	**bitflips regs**	-0.0000	0.1229	0.0157	0.9183
	bitflips memory	-0.0000	0.1466	0.0190	0.8798
	all features	-0.0000	0.1152	0.0140	0.9354
	only single instr	-0.0000	0.1581	0.0235	0.8167
	basic ilpm	-0.0000	0.1467	0.0191	0.8797
	branch taken	-0.0000	0.1369	0.0183	0.8890
	bitflips bus	-0.0000	0.1163	0.0140	0.9351
CPU	**bitflips regs**	-0.0005	4.3411	0.8906	0.9047
	bitflips memory	-0.0006	5.8394	0.9945	0.8811
	all features	-0.0003	4.2284	0.8044	0.9222
	only single instr	-0.0001	6.8246	1.4697	0.7404
	basic ilpm	-0.0006	5.8336	0.9941	0.8812
	branch taken	-0.0006	5.7050	0.9922	0.8817
	bitflips bus	-0.0003	4.2393	0.8167	0.9198

Table C.2: Statistics for the coremark benchmark.

domain	model	$\bar{\epsilon_r}$ (%)	$\max\{\lvert\epsilon_r\rvert\}$ (%)	RMS(ϵ_r) (%)	R^2
ULL RAM	**bitflips regs**	0.4717	20.3224	4.2428	-0.3254
	bitflips memory	-0.2713	21.9857	4.4179	-0.4371
	all features	-2.0403	22.3501	4.7963	-0.6938
	only single instr	-0.1930	21.8069	4.2727	-0.3442
	basic ilpm	-0.2692	21.9886	4.4187	-0.4376
	branch taken	-0.2398	21.8899	4.2817	-0.3498
	bitflips bus	-2.4287	23.1898	5.1692	-0.9674
ULP RAM	**bitflips regs**	-0.0512	0.1568	0.0552	-7.8659
	bitflips memory	-0.0675	0.1654	0.0701	-13.3241
	all features	-0.0335	0.1433	0.0420	-4.1282
	only single instr	-0.0569	0.1907	0.0623	-10.2929
	basic ilpm	-0.0676	0.1652	0.0702	-13.3340
	branch taken	-0.0681	0.1647	0.0710	-13.6750
	bitflips bus	-0.0323	0.1445	0.0418	-4.0801
CPU	**bitflips regs**	-4.3165	8.2042	4.4486	-14.7143
	bitflips memory	-4.9544	9.2453	5.0928	-19.5950
	all features	-3.5202	7.6175	3.7093	-9.9254
	only single instr	-4.4539	8.3754	4.6131	-15.8975
	basic ilpm	-4.9524	9.2549	5.0906	-19.5770
	branch taken	-4.9567	9.3351	5.0987	-19.6426
	bitflips bus	-3.6528	7.6699	3.8193	-10.5830

Table C.3: Statistics for the fib benchmark.

domain	model	$\bar{\epsilon_r}$ (%)	$\max\{\lvert\epsilon_r\rvert\}$ (%)	RMS(ϵ_r) (%)	R^2
ULL RAM	**bitflips regs**	4.2100	6.9835	4.2708	-76.1843
	bitflips memory	4.4448	6.9589	4.4977	-84.6041
	all features	4.2791	8.1959	4.3767	-80.0598
	only single instr	4.1576	6.4722	4.2064	-73.8720
	basic ilpm	4.4690	6.9922	4.5219	-85.5278
	branch taken	3.7356	6.1427	3.7913	-59.8238
	bitflips bus	5.6040	9.7388	5.6915	-136.0771
ULP RAM	**bitflips regs**	-0.0870	0.1050	0.0874	-254.5757
	bitflips memory	-0.0808	0.1053	0.0812	-219.4866
	all features	-0.0866	0.1152	0.0870	-252.5973
	only single instr	-0.0778	0.1030	0.0782	-203.7241
	basic ilpm	-0.0811	0.1056	0.0815	-221.1138
	branch taken	-0.0661	0.0879	0.0664	-146.5622
	bitflips bus	-0.0901	0.1199	0.0906	-273.5057
CPU	**bitflips regs**	-2.1054	3.0939	2.1204	-271.3986
	bitflips memory	-1.8953	3.1444	1.9168	-221.6031
	all features	-2.5240	3.9544	2.5474	-392.1537
	only single instr	-2.4881	3.6569	2.5065	-379.6310
	basic ilpm	-1.8702	3.1166	1.8915	-215.7632
	branch taken	-1.7558	2.9822	1.7774	-190.3940
	bitflips bus	-2.1379	3.4595	2.1601	-281.7127

Table C.4: Statistics for the fir benchmark.

domain	model	$\bar{\epsilon_r}$ (%)	$\max\{\lvert\epsilon_r\rvert\}$ (%)	RMS(ϵ_r) (%)	R^2
ULL RAM	**bitflips regs**	5.3796	7.0859	5.5926	-34.6815
	bitflips memory	5.1645	6.9956	5.3741	-31.9482
	all features	4.7244	5.9814	4.9132	-26.5389
	only single instr	4.9742	6.2366	5.1722	-29.5189
	basic ilpm	5.1709	7.0177	5.3809	-32.0314
	branch taken	4.6767	6.1424	4.8663	-26.0148
	bitflips bus	5.5573	7.1411	5.7804	-37.1182
ULP RAM	**bitflips regs**	-0.0592	0.0703	0.0614	-678.4446
	bitflips memory	-0.0638	0.0716	0.0661	-787.2132
	all features	-0.0699	0.0786	0.0724	-944.9627
	only single instr	-0.0630	0.0721	0.0653	-768.5090
	basic ilpm	-0.0639	0.0718	0.0662	-789.0953
	branch taken	-0.0541	0.0616	0.0561	-566.3646
	bitflips bus	-0.0730	0.0823	0.0757	-inf
CPU	**bitflips regs**	-0.6485	0.8501	0.6820	-19.4602
	bitflips memory	-0.8397	1.1586	0.8908	-33.9088
	all features	-1.4796	1.6645	1.5349	-102.6362
	only single instr	-1.6349	1.8528	1.6968	-125.6494
	basic ilpm	-0.8331	1.1564	0.8848	-33.4389
	branch taken	-0.7581	1.0927	0.8117	-27.9824
	bitflips bus	-1.1586	1.3519	1.2049	-62.8639

Table C.5: Statistics for the hanoi benchmark.

domain	model	$\bar{\epsilon_r}$ (%)	$\max\{|\epsilon_r|\}$ (%)	RMS(ϵ_r) (%)	R^2
ULL RAM	**bitflips regs**	4.4854	6.0249	5.0454	-55.4978
	bitflips memory	4.7446	6.3009	5.3304	-62.0596
	all features	3.2197	4.6494	3.6498	-28.5647
	only single instr	4.4914	5.9959	5.0509	-55.6192
	basic ilpm	4.7787	6.3401	5.3683	-62.9595
	branch taken	4.0157	5.5109	4.5236	-44.4149
	bitflips bus	4.6188	6.1739	5.1946	-58.8883
ULP RAM	**bitflips regs**	-0.0651	0.0791	0.0725	-inf
	bitflips memory	-0.0577	0.0697	0.0643	-inf
	all features	-0.0662	0.0803	0.0738	-inf
	only single instr	-0.0530	0.0656	0.0591	-inf
	basic ilpm	-0.0581	0.0702	0.0648	-inf
	branch taken	-0.0420	0.0510	0.0468	-741.3051
	bitflips bus	-0.0706	0.0851	0.0787	-inf
CPU	**bitflips regs**	-0.7938	1.0991	0.8875	-72.0664
	bitflips memory	-0.5535	0.9266	0.6264	-35.3908
	all features	-1.4779	1.9196	1.6495	-251.3906
	only single instr	-1.1832	1.5102	1.3201	-160.6486
	basic ilpm	-0.5164	0.8795	0.5853	-30.7734
	branch taken	-0.3926	0.7282	0.4494	-17.7367
	bitflips bus	-1.0173	1.3553	1.1362	-118.7579

Table C.6: Statistics for the quicksort benchmark.

domain	model	$\bar{\epsilon_r}$ (%)	$\max\{\lvert\epsilon_r\rvert\}$ (%)	$\mathrm{RMS}(\epsilon_r)$ (%)	R^2
ULL RAM	**bitflips regs**	5.7548	7.3097	6.1597	-24.7366
	bitflips memory	6.0414	7.5484	6.4599	-27.3061
	all features	5.5821	7.8184	6.0177	-23.5637
	only single instr	5.0719	6.4091	5.4245	-18.9596
	basic ilpm	6.0442	7.5441	6.4625	-27.3285
	branch taken	5.3084	6.9120	5.6894	-20.9567
	bitflips bus	6.9484	9.0241	7.4435	-36.5824
ULP RAM	**bitflips regs**	-0.0656	0.0851	0.0700	-94.6798
	bitflips memory	-0.0590	0.0761	0.0630	-76.4343
	all features	-0.0741	0.0869	0.0789	-120.7666
	only single instr	-0.0685	0.0855	0.0731	-103.5436
	basic ilpm	-0.0590	0.0763	0.0630	-76.5320
	branch taken	-0.0441	0.0566	0.0472	-42.4775
	bitflips bus	-0.0775	0.0898	0.0826	-132.1747
CPU	**bitflips regs**	-0.4209	1.1070	0.5127	-0.2516
	bitflips memory	-0.1653	0.9943	0.3557	0.3976
	all features	-1.2004	2.0498	1.3029	-7.0839
	only single instr	-1.9927	3.1083	2.1555	-21.1237
	basic ilpm	-0.1625	0.9965	0.3548	0.4006
	branch taken	-0.0498	0.9005	0.3225	0.5047
	bitflips bus	-0.8051	1.7085	0.9058	-2.9071

Table C.7: Statistics for the tak benchmark.

domain	model	$\bar{\epsilon_r}$ (%)	$\max\{\lvert\epsilon_r\rvert\}$ (%)	RMS(ϵ_r) (%)	R^2
ULL RAM	**bitflips regs**	3.2539	5.1866	3.4341	-54.9326
	bitflips memory	3.3194	5.2441	3.5005	-57.1186
	all features	1.5497	3.6780	1.8188	-14.6895
	only single instr	3.2650	4.9917	3.4259	-54.6672
	basic ilpm	3.3464	5.2802	3.5280	-58.0331
	branch taken	2.4877	4.3175	2.6640	-32.6609
	bitflips bus	3.0164	5.3268	3.2409	-48.8158
ULP RAM	**bitflips regs**	-0.0658	0.0733	0.0679	-548.9919
	bitflips memory	-0.0633	0.0703	0.0652	-506.9827
	all features	-0.0639	0.0709	0.0659	-517.0165
	only single instr	-0.0624	0.0696	0.0644	-493.6122
	basic ilpm	-0.0636	0.0705	0.0656	-512.5507
	branch taken	-0.0453	0.0522	0.0467	-259.5857
	bitflips bus	-0.0685	0.0756	0.0706	-593.3653
CPU	**bitflips regs**	-1.0024	1.3579	1.0697	-8.5393
	bitflips memory	-0.9437	1.3190	1.0132	-7.5576
	all features	-1.5872	1.8783	1.6497	-21.6854
	only single instr	-1.4486	1.8655	1.5271	-18.4395
	basic ilpm	-0.9140	1.2914	0.9846	-7.0814
	branch taken	-0.7733	1.1512	0.8483	-4.9983
	bitflips bus	-1.1013	1.3977	1.1588	-10.1941

Bibliography

[ACPS01] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Sarta. An instruction-level power analysis model with data dependency. *VLSI Design*, 2001:245–273, 2001.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power cmos digital design. *IEICE Transactions on Electronics*, 27(4):371–382, 1992.

[DLL+17] Chenchen Deng, Leibo Liu, Yang Liu, Shouyi Yin, and Shaojun Wei. Pmcc: Fast and accurate system-level power modeling for processors on heterogeneous soc. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 64:540–544, 2017.

[EEM22] EEMBC. Coremark. `https://www.eembc.org/coremark`, 2022. Accessed on 2022-7-7.

[Eik21] Fridtjof Gerdsønn Eikanger. Instruction-level power modeling. Master's thesis, Norwegian University of Science and Technology, December 2021.

[GVH00] Tony Givargis, Frank Vahid, and Jörg Henkel. A hybrid approach for core-based system-level power modeling. *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 141–145, 2000.

[Ini22a] Accellera Systems Initiative. SystemC the language for system-level design, modeling and verification. `https://systemc.org/about/systemc/overview/`, 2022. Accessed on 2022-7-3.

[Ini22b] Accellera Systems Initiative. SystemC transaction level modeling (tlm). `https://systemc.org/about/systemc/tlm/`, 2022. Accessed on 2022-7-3.

[Int21] RISC-V International. History of RISC-V. `https://riscv.org/about/history/`, 2021. Accessed on 2021-12-17.

[IZZH19] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. Copper: Soft real-time application performance using hardware power capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 31–41, 2019.

[NKL+03] Spiridon Nikolaidis, Nikolaos Kavvadias, Theodore Laopoulos, Labros Bisdounis, and Spyros Blionas. Instruction level energy modeling for pipelined processors. *J. Embed. Comput.*, 1:317–324, 2003.

[RJ98] Jeffry T. Russell and Margarida F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, pages 328–333, 1998.

[SWN+20] Sivert T. Sliper, William Wang, Nikos Nikoleris, Alex S. Weddell, and Geoff V. Merrett. Fused: Closed-loop performance and energy simulation of embedded systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 263–272, 2020.

[Tho98] Scott Thompson. Mos scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, 1998.

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration Systems*, 2(4):437–445, 1994.

[Uni19] University of California, Berkeley. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, 2019.

[Vil19] Luis David López Tello Villafuerte. System-level power modeling for automotive asics. Master's thesis, Technische Universität Kaiserslautern, October 2019.

[Wan17] Wei Wang. *An improved instruction-level power and energy model for RISC microprocessors*. PhD thesis, University of Southampton, 2017.

[WWDS94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, 1994.