POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

A feasibility study for an energy prediction and optimization framework

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Alessandro Bertulli

Advisor: Prof. Giovanni Agosta

Co-advisor: Daniele Cattaneo

Academic year: 2022-2023

1. Introduction

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).
By estimating the amount of energy required, developers can better estimate the resources needed 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.
Usually, software is optimized with respect to the time required to perform its work. However, time optimizations do not imply that the power consumption is contextually reduced. For instance, there can theoretically be optimizations that reduce the software time cost, but increase the power consumption. We hypothesize instead a compiler driver that can specifically operate energy aware optimizations. Such tool would receive as inputs the source code we'd like to compile, along with a flag indicating 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 that we assess during our work, and in the end we produce an introductory feasibility analysis of the construction of such a tool.
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 may use 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; therefore, we need to extract such information (and thus, the model) from the actual manufactured chip, keeping our analysis as black box as possible.
In this work we focus our analysis only on the first part of this tool, that is, the building of the oracle.

2. State of the art

2.1. Choosing the estimation technique

To search for the best estimation technique, we took into consideration different methods.
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 [6]. The simulator receives as input a description of the device in a suitable hardware description language (HDL), possibly (but not necessarily) Verilog [7] or VHDL [8], along with some code to execute on its physical counterpart, and it simulates the processor's working (up to the simplified physical effects). This is however not feasible for us, since we aim to build the oracle even when we cannot access the internal workings of the device under test (DUT).
Another approach is to rely on tools provided directly by the manufacturers, such as hardware performance counters (HPCs) and performance monitor counters (PMCs) [6]. Such tools are exposed 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 [4], with errors up to 32 %, so they are not a very sound approach. We considered this error too high to guide a reliable optimization toolchain.
In the end, we resolved to use direct current readings to build an ISA model of the processor. They offer great accuracy, can be relatively easy to build even when the disclosed information is limited, and we can benefit from a great theoretical background to build our work on.

2.2. Choosing the ISA model

A common starting point when studying power consumption is [14]. There, the authors outline an idealized model for the energy consumed by a program. This model is theoretically extremely precise (with errors as low as 0.26 %), but practically useless on its own, since it uses a set of parameters related to the power dissipated by each CPU activity. To use it we need first to accurately estimate all the parameters, which is the major difficulty of the experimental analysis.
Lee et al. in [10] 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. The key idea here is that their model uses arbitrary functions of the code profiled. By choosing the correct function, we can analyse a specific behaviour or effect of the code under test. The authors list some of them: we took inspiration and implemented some of our own, as explained later.
A completely different approach is proposed in [9], where the authors try to model a power consumption score in a purely statistical way. They observe that the switching activity of the internal signals of a processor is proportional to the power consumed. Starting by this, they observe how signals vary in a realistic setting, dividing them in two parts: one more static (related to the most significant bits) and one more dynamic (related to the least significant bits). This approach is interesting and achieves low error rates, but it requires a still too deep knowledge of the processor internals. For this reason, we discard this method and we focus ourselves on linear models.
The last contribution we can analyse is provided by Georgiou et al. in [5]. There, the authors employ a linear model that separates the power

consumption into three contributions: a common power equal for all instructions, an instruction specific power, and a switching overhead related to the interleaving of instructions. The results are good, as the average absolute error is about 3.2 %, and the worst is around 10 %. The authors use a particular processor and leverage some of its peculiar characteristics, so the work can not be easily ported; however, we can use some of its basic principles to create a more general model. In practice, we combine ideas from [5], [10], [14] to create a model that is as general purpose as possible.

3. Model architecture and construction

3.1. The physical tools

To build an example oracle, we chose to profile 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) [13], and is easily programmable by end users using STMicroelectronics' tools. The MCU is hosted on a STM32 Discovery kit board.[b]
Moreover, to measure the power absorbed by the CPU we used the Qoitech™ Otii Arc Pro, an easy to use digital voltmeter capable of measuring time varying tension with a decent precision. Since the device actually measures only tension, the current reading is obtained by using a shunt resistor, connected in series with the processor. The apparatus is placed inside a Faraday's box to shield it from electromagnetic disturbances.
The Otii Arc Pro comes with a graphical user interface (GUI) program delivered by the manufacturer, from which it is possible to start and stop a recording and export its data in a CSV file. Using a GUI can be a problem as the time to save a measurement will considerably be slower than using a command line tool. Since the machine used in this work was running the X11 protocol, we picked the `xdotool` software,[c] to automate the physical recording process.

3.2. The measurement workflow

The overall process is handled by a script written in Emacs Lisp [11], that invokes the other external programs, script and functions to run a measurement battery.
First, a small library of Emacs Lisp functions we wrote generates the instructions to test starting from a given template. For instance, we can run the command

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

to produce

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

It substitutes each placeholder (`%1`, `%2` ecc.) with the corresponding element from the provided ranges, building their Cartesian product. The ranges can either be a sequence of strings or a numeric range in the form `start:step:end`.
Then, each instruction in injected into the C code to be flashed onto the STM32 board. The official STMicroelectronics editor provides a firmware and a C template that contains a `main` function with an infinite loop. By placing the instruction into that, wrapped in an `__asm__` directive, we can make the CPU execute that instruction indefinitely. The resulting C file is compiled and linked with the STMicroelectronics firmware: to do so, we used GCC [12].
We note that collecting power readings from arbitrary memory instructions is difficult, as the board may implement memory protection. 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.
The compiler outputs a binary image, that can be loaded onto the board with the `st-flash` program (also provided by STMicroelectronics). 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

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

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.

3.3. The model

We model the power of a sequence of instructions as

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

For each instruction i we compute the power consumed by the instruction (P_i) and we scale it by an overhead multiplicative factor that depends on the current and the next instructions ($O_{i,i+1}$). Then we compute the weighted sum of the resulting values with respect to the clock cycles occupied by the instructions. The terms P_i and $O_{i,i+1}$ are estimated by multiple linear regression, where we take as explanatory variables the results of some functions of the instructions under test. Such variables can be quantitative or categorical. For the term P_i, they encode the instruction mnemonic opcode, whether the current instruction updates the application program status register (APSR), whether it uses conditional execution, whether it uses the barrel shift for its second operand (and by how much), whether the second operand is an immediate, whether the destination register of the instruction is equal to one of the source registers, and the binary weight of its binary representation.
Some variables are correlated with each other, as for instance we expect some of their effect to be dependent on the instruction mnemonic of the instruction. We cannot measure all the possible combinations of effects, as it would require prohibitive amounts of time, so we limit ourselves to only some of them. Doing so means that the model is not properly fitted for all the instructions, and trying to predict the new ones may lead to runtime errors. So, we provide a second, simpler model, to be queried if the first one fails.
In a similar way, we estimate the inter instruction overhead $O_{i,i+1}$, using as explanatory variables the mean predicted power of the two instructions under test, their Hamming distance, their binary weights and their mnemonic opcodes. Again, some of the variables are correlated, and a second simpler model is provided. Data for this model is collected similarly with respect to the single instructions, but this time we tested a loop of two instructions. For both the power and the overhead estimations, a fallback constant value is provided in case even the second model fails.
The final oracle (written in the Julia programming language [2]) needs a sequence of instructions to analyse. We explored two possibilities: first, we can simulate the execution of the program with an emulator like QEMU [1], estimating the particular interleaving of instructions; second, we can simply read the disassembled binary code of the application and estimate all the basic blocks of code. We note that the former can be done only if the program is completely deterministic (i.e., it does not require external inputs), and the emulator does not support all the architectures.

4. Experimental results

4.1. Data exploration

When deciding which regression variables to use in the linear models, we searched for patterns in the collected data during the measurement process. We included all the variables of the single instruction model that have an impact on the power consumption, and that lead to good predictions. The tentative explanation we give for these effects is that different capabilities of the ARM ISA are executed by different areas of the processor, and activating them means changing the total number of logic gates involved in the calculation. However, their impact is not constant, meaning that different instructions act differently with respect to the same effect, and for some of them the power difference is even reversed in sign. Therefore, we should correlate all the effects with the instruction mnemonic opcode whenever is possible, as we did in the single instruction complete model.

4.2. Goodness of the models

Since our predictor is composed of different linear models, assessing their single goodness does not completely capture the adequacy of the oracle; however, computing some basic metrics can give us an intuition about it. We computed the R^2 index, the mean absolute percentage error (MAPE), both with the train data and in a 10-fold cross validation setting (we report the average value), and the percentage of parameters for which the common hypothesis test $\mathbb{H}_0 : \beta_i = 0$

versus $\mathbb{H}_1 : \beta_i \neq 0$ presents a p-value higher than the usual level of significance of 0.05. The results are shown in Table 1 at page 6. For the inter-instruction overhead models we collected fewer data points and the errors are very much higher; however, they still perform reasonably well when validating the predictor with real programs.

4.3. Predictor validation

The real validation can be done when testing the predictor against real programs. First, we wrote five simple programs that can be tested in a totally deterministic way, to evaluate the model performance with the QEMU generated trace. Such programs compute the nth Fibonacci number (both recursively and iteratively), the factorial of a natural number n (recursively and iteratively), and the sum of the first n integers. Then, to test a more general program, and to assess the performance against a real compiler optimization, we used a series of benchmarks employing the Taffo framework [3]. We did in total three validations: the deterministic programs with the QEMU trace, the Taffo benchmarks with the normal trace, and the deterministic programs with the normal trace.
For each of these programs, we wrote the code on the board and measured the power consumption. Then, we used our predictor to produce an estimate of the consumption. We then produced a matrix of the pairwise power differences between each couple of programs in the same group, both for the measured and for the predicted powers. Since we are only interested in knowing if a compiler modification of the code is saving energy or not, we consider the predictor successful if the difference in measured power and the difference in predicted power have the same sign. The results are plotted in Figure 4.1: we can see that the trace obtained with QEMU performed badly, whereas the Taffo programs with the normal trace gave much better results. In particular, if we take each Taffo program, and check only the prediction regarding the optimization of that single benchmark, the rate of success raises up to 86.21 %. This is an interesting result because it's the closest one to what we expect will be the predictions in an optimization toolchain. Lastly, we see that using the normal trace with the deterministic programs improves the results also for them.

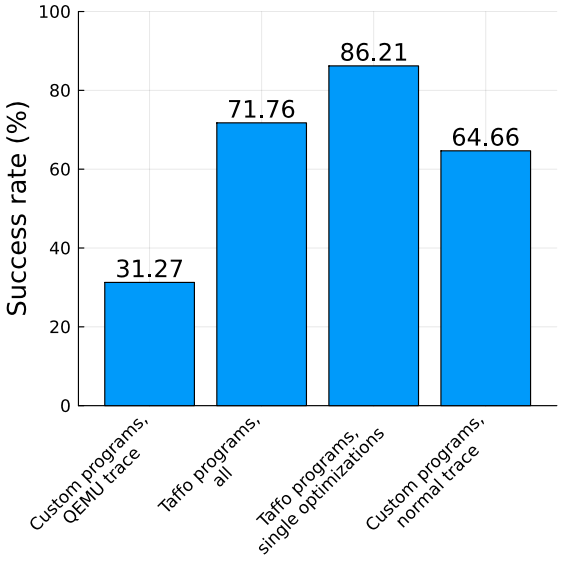

Figure 4.1: Success rate for the oracle, with different benchmark suites.

5. Feasibility study

Our example oracle is able to correctly predict the effect of a compiler optimization in more than half of the cases. Elaborating on this work, we outline a series of requirements to build similar oracles in a general purpose way.

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

Table 1: Goodness indexes for the single instruction linear models.

Model	R^2	MAPE (%)	mean MAPE 10-fold (%)	Parameters with p-values > 0.05 (%)
Single instr., complete	0.7938	6.4	6.6	26.92
Single instr., reduced	0.7711	7.27	7.41	53.88
Inter-instr., complete	0.9302	1051.25	1064.35	74.22
Inter-instr., reduced	0.4225	1097.69	1098.48	25.0

representations that mask the register allocations, or the final ISA mapping (such as, for instance, the LLVM intermediate representation (LLVM-IR)), couldn't have predicted these effects.

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.

References

[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] 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.

[5] 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.

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

[7] "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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.