https://doi.org/10.1007/s42979-020-00405-7

ORIGINAL RESEARCH

A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis

Charalampos Marantos[1] · Konstantinos Salapas[1] · Lazaros Papadopoulos[1] · Dimitrios Soudris[1]

Abstract
The design requirements of modern applications that target embedded systems, such as the need for high performance and low energy consumption, impose challenges on developers. Software tools capable of providing performance and energy consumption estimations are useful for addressing these challenges. Such tools aim to reduce development time and alleviate the time-to-market pressure. In this work, we propose a flexible tool that enables the estimation of performance and energy consumption of the application on embedded devices, providing a complete methodology based on which the user can add estimation models for various platforms. In contrast to existing tools that either rely on dynamic instrumentation or require detailed modeling of the hardware, the proposed tool leverages static analysis techniques applied at instruction level coupled with data-driven regression models. The proposed method is tested using a widely used benchmark suite for evaluation.

Keywords Energy consumption · Performance · Static analysis · Estimation

Introduction

The number of battery-driven IoT devices is expected to reach 35 billions in 2025.[1]. Nowadays, connected devices are installed in industrial systems for improving productivity, in healthcare systems and wearables for providing valuable feedback, as well as in various others environments. AI chips integrated in consumer devices, such as high-end smartphones, tablets, smart speakers, and wearables, further increase the capabilities of edge devices of IoT networks.[2] Indeed, there are increased expectations on both the research community and the industry for delivering advanced technological achievements that will enable new IoT applications.

However, there are several challenges that IoT application developers face. According to a recent IoT-oriented embedded market study,[3] application performance and energy efficiency are some of the most critical issues that IoT developers are expected to address. Indeed, meeting the energy requirements of IoT applications, as well as the performance and security requirements is often very challenging.

Performance and energy efficiency in embedded devices can be addressed at various levels. Improvements at hardware level include the introduction of fast, but ultralow-power embedded CPUs and application-specific accelerators (e.g., neural compute units[4]), which typically achieve significant energy savings in various IoT application scenarios and meet performance constraints. Other approaches include energy-efficient wireless interfaces, such as the Bluetooth Low Energy. Energy harvesting has been proposed for IoT

This article is part of the topical collection "Interaction between Energy Consumption, Quality of Service, Reliability and Security, Maintainability of Computer Systems and Network" guest edited by Erol Gelenbe.

✉ Charalampos Marantos
hmarantos@microlab.ntua.gr

Konstantinos Salapas
ksalapas@microlab.ntua.gr

Lazaros Papadopoulos
lpapadop@microlab.ntua.gr

Dimitrios Soudris
dsoudris@microlab.ntua.gr

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece

[1] https://www.helpnetsecurity.com/2019/05/23/connected-devices-growth/.

[2] https://www.eetimes.com/putting-ai-into-the-edge-is-a-no-brainer-and-heres-why/.

[3] https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[4] https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html.

applications in which conventional power sources are not available. Finally, improving local computations at the edge of IoT networks, so that the amount of data need to wirelessly be transmitted to the Cloud is minimized, is a typical way to improve both performance and energy efficiency of IoT applications.

There are various works in the embedded system domain that highlights the importance of energy-efficient application software that meets performance constraints. Inefficient software can drive low-power hardware to waste the system's energy budget, regardless of the application performance [8]. Relevant works in the literature focus on source-to-source optimizations at application level for increasing application performance and energy efficiency by improving the data flow and memory utilization [10, 15].

From software engineering perspective, there are several works that promote the use of tools and methods to allow energy consumption to be an important software design goal, by retaining performance requirements. They propose best practices and guidelines for all phases of software development lifecycle based on empirical evaluation [17]. Other works, also based on empirical evaluation, try to determine the impact of typical source code refactorings on energy consumption [9].

It is widely accepted that performance and energy efficiency are important qualities for software. However, there are a limited number of tools available that assist software developers on evaluating the software from application performance and energy consumption perspective [16, 17]. Developers need to monitor the energy consumption of the software during the development phase instead of measuring energy after deployment [5].

Although measuring performance and energy consumption on the actual hardware is potentially the most accurate approach, there are several limitations. Hardware is often not accessible, which may involve sophisticated equipment and special hardware knowledge. As a result, due to the lack of tools in the current state of the art, development costs for energy-efficient systems are higher than for energy-wasteful systems due to the extra effort required to take energy consumption into account [5].

An alternative approach, which is more accessible to application developers, is the program independent and hardware-specific performance and energy models. They associate basic software constructs (source code blocks and basic blocks in the intermediate representation) with energy consumption [5]. Such models can be implemented at various levels, such as at IR or at instruction level.

This manuscript proposes a holistic design methodology (framework) for enabling the estimation of performance and energy consumption of applications on various devices through static analysis. In contrast to relevant approaches, the proposed tool is based on simple features retrieved from instruction level but generic enough to support many architectures and instruction sets. Furthermore, experimental results highlight the ability of the proposed mechanism to achieve acceptable results, without the necessity of accurate hardware modeling, which in turn leads to a plug and play solution. Additionally, the work described in this manuscript offers a methodology for the user to easily add new embedded devices in the framework.

The method proposed in this manuscript is part of the SDK4ED project, funded by the European Union's Horizon 2020 research and innovation programme. The goal of SDK4ED is to minimize the cost and the development time of software development processes, by providing tools for automatic optimization of multiple quality requirements, such as technical debt, energy efficiency, dependability, and performance. The SDK4ED project so far has proposed numerous research and technical methods either with respect to the forecast and optimization of the targeted quality attributes (Energy, Maintainability, and Dependability) [4, 7, 18, 21], or by investigating and studying the trade-offs between them [14, 19, 20]. The research work presented in this article is part of the energy consumption estimation component, which is integrated in the energy toolbox of the SDK4ED platform. The proposed framework aims at:

- Avoiding the limitations of dynamic instrumentation techniques, such as the execution time overhead and enabling an estimation of the consumed energy by applications in a fast, convenient, and user-friendly way.
- Using static analysis techniques, so that the energy estimation component uses similar approaches with the rest of the SDK4ED toolboxes (TD and dependability) that rely on static analysis techniques entirely.
- Enabling the estimation of energy consumption on various architectures. Apart from the architectures already supported in the current version of the framework, the proposed methodology offers users the capability of adding estimation models for other devices.

The rest of the manuscript is summarized as follows: Section "Related Work" provides an overview of relevant approaches found in the literature, while the proposed framework, as well as its components, are discussed in detail in Section "Proposed Methodology". The efficiency of the introduced solution in well-known embedded devices is evaluated in Section "Evaluation". Finally, Section "Conclusion" concludes the manuscript.

Related Work

Initial approaches towards performance predictions at instruction level include works based on simple machine learning algorithms (i.e., regression) [1]. This work adopts a measurement-based method and achieves very promising results. However, it targets only on microcontrollers and it is based on a specific Instruction Set. Furthermore, this approach uses a very limited dataset (60 test programs).

More recent approaches that provide increased accuracy utilize machine learning techniques for cross-platform performance and power predictions [22, 23]. These approaches are based on dynamic instrumentation and include dynamic features in the proposed estimation models such as cache misses, cycles, etc. Although this procedure leads to very accurate estimations, it imposes a lot of problems for being a part of a Software Development Toolbox. The dynamic instrumentation not only requires application execution, but also adds a large time overhead. Furthermore, a lot of manual actions are needed by the developer (such as adding annotations in the source code).

Mira [13] leverages the ROSE compiler and estimates fp operations per block based on user input (i.e., number of loop iterations) and an analytical hardware architecture description file. This tool does not estimate performance or energy consumption, but it is worth referring as it gave us a lot of ideas for designing the proposed methodology, such as the analysis of the Abstract Syntax Tree, the focus on the loops, as well as the requirement of some dynamic information (such as the number of loop iterations) by the user.

Another category of estimation tools includes the Worst-Case execution Time tools (WCT) [6, 11]. These tools aim to characterize and estimate the applications worst-case performance based on an iterative approach without having any information about the input. The usual criticism to this kind of tools includes the following facts: they are estremely slow, needing even days for performing the estimations, they are not so accurate, and they can support only specific architecture models. Approaches that aim to provide improvements are also proposed [2].

Finally, another kind of tools that can be used to predict performance, in terms of throughput or number of cycles, include Ithemal [12], LLMV-mca[5] and Intel Architecture Code Analyser (IACA).[6] Ithemal is based on machine learning techniques (an LSTM RNN), while LLMV-mca and Intel IACA model x86-64 architectures. These tools are considered very useful as they give as an output the estimated throughput of executing a code basic block on a specific architecture. In the work introduced in this manuscript, the information retrieved by these tools is also combined with additional features, being an input in the proposed performance and energy consumption estimation models.

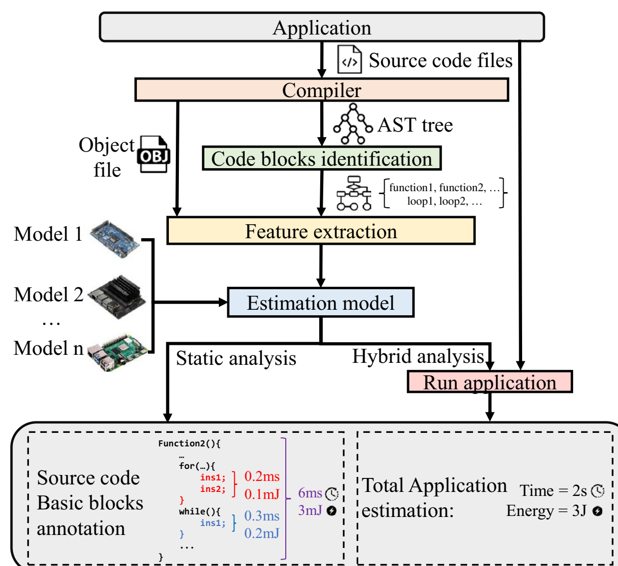

Fig. 1 Overview of the proposed methodology

Proposed Methodology

This section describes in detail the proposed methodology for estimating performance and energy in embedded platforms through static analysis. The proposed flow is depicted in Fig. 1. The input of the methodology is the application source code (written in C/C++) and the output is the predicted execution time, as well as the estimated energy consumption.

The introduced mechanism consists of four steps depicted in Fig. 1. Initially, the application is compiled. The Abstract Syntax tree (AST) produced by the compiler front end (in the context of the proposed methodology, CLANG was utilized) is forwarded to the Code blocks identification step. This step analyses the AST tree and identifies functions and loop statements (e.g., *for*, *while*), splitting the application source code into corresponding blocks (e.g., loop body, function body, etc.).

The object file generated by the compiler is given as an input to the next step, which is responsible for extracting the features that will be used by the estimation model (next step) to predict the performance and energy consumption of running the application on various embedded systems. This step takes as an input the targeted embedded device and estimates the time and energy required for executing each application code block through the device-specific model. The output

[5] https://llvm.org/docs/CommandGuide/llvm-mca.html.

[6] https://software.intel.com/content/www/us/en/develop/articles/intel-architecture-code-analyzer-download.html.

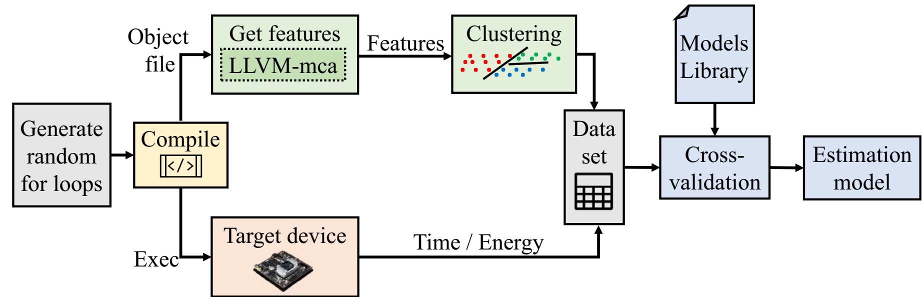

Fig. 2 Proposed procedure for building device estimation models

of the estimation model, regarding each code block, is presented by the proposed tool. Results for executing the total application are also provided. It is worth mentioning that, especially regarding the loops, a dynamic information (e.g., the number of iterations) is considered necessary. The proposed framework supports both gathering this information as an input from the user to perform only static analysis, or executing the application to retrieve this information and to combine the individual code blocks information in a proper way that corresponds to the actual execution of the total application.

Building the Estimation Models

In this subsection, we discuss the proposed procedure to build estimation models capable of predicting the performance and the energy consumption of the applications being executed on various embedded devices. Figure 2 depicts the introduced methodology, which is described in a detailed manner in the rest of the Section.

Features

As the proposed solution aims to be cross-platform and cross-architecture, the introduced solution proposes the generation of general features that model and characterize the application's behavior and characteristics that directly affect the performance and energy consumption. Features have to model the application's behavior and computational requirements. They are based on the application's assembly code as well as the output of the LLVM-mca tool analysis. The assembly instructions are categorized, building a small number of generalized features in order for the proposed method to be generic enough and applicable to different architectures and instruction sets. Instructions belonging to the same category have similar execution time. The selected features are given below. The rest of the section describes in detail the methodology followed for selecting these features:

- number of instructions (INS)
- estimated throughput (by LLVM-mca)
- number of LOAD instructions
- number of STORE instructions
- number of OP (operations) instructions
- number of instructions in INS class 1 (add, sub, shift, and mul)
- number of instructions in INS class 2 class (conv, arrays, div)
- OP, LOAD, and STORE instructions order.

The LOAD and STORE operations have an important impact on the program execution in terms of time and energy as they require access in the system's memory (either the CPU cache or the main memory). The average time overhead of these operations as measured on an ARM Cortex-A57-based device (Nvidia Tegra TX1) is from 2.2×10^{-7} up to 4.4×10^{-7} ms for STORE and from 1.2×10^{-7} up to 3.2×10^{-7} ms for LOAD operations. The time variations depend on the type of data, the size, as well as the cache behavior. The OP (operation) type instructions are classified into two more categories. The first category includes the *add*, *sub*, *shift*, and *mul* instructions. These instructions are the majority of the operations in all the applications that have been analyzed in the context of this manuscript. *Shift* operations seem to have similar time overhead with the addition operations. On the other hand, *mul* operations seem to have lager overhead (as they include multiple additions). For example, in ARM-Cortex A57, the *add, sub*, and *shift* operations time overhead does not exceed 2.2×10^{-7} ms, while *mul* operation execution time is up to 2.4×10^{-7} ms. However, the overhead of the *mul* operations is not considered significantly larger leading us to include them also in the same category.

Most of the applications use array operations. Arrays processing is considered very important for the total application's performance and energy consumption. Array operations are also coupled with memory operations. For example, when an array element if accessed for the first time, we have a cache miss and the time and energy needed for the transaction of the data from the main memory is very important. If the specific block is already in the cache memory, the overhead is relatively smaller. For the purpose of the proposed methodology and due to the fact that we focus on static source code analysis only, we

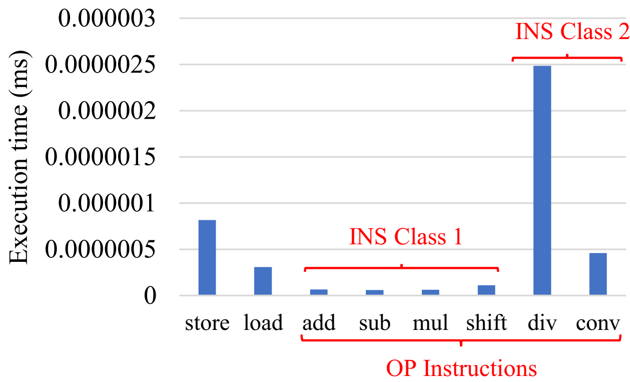

Fig. 3 Average execution time of instruction classes on two ARM processors

assume that the cache memory is ideal and we do not take into account any hardware information about the cache architecture. However, this impact of array operations is measured indirectly by including all the instructions that perform array operations into a second category. Usually, especially in 64-bit systems, when an array operation is performed, the compiler also produces conversion instructions (*conv*) (such as *cltq, cdqe, and movslq* in *x*86 assembly) to convert the index value from a 32 to 64 bits for the pointer to access the 64-bit memory addresses. *Div* operations use pipeline and include multiple stages, leading to increased time and energy overhead. The array operations and the conversion time overhead in ARM-Cortex A-57 varies from 2×10^{-7} ms up to 1.3×10^{-6} ms, while division operations sometimes need up to 1.6×10^{-6} ms to be executed.

The most important of the rest of the operations (included only in the OP instruction feature) are *jmp* operations, comparison operations *cmp*, as well as *mov* instructions. The *jmp* and *cmp* operations are not present in the basic blocks analyzed by our tool (loop bodies and function bodies) as they are connected with changes in the program flow. As stated above, due to the fact that the proposed tool is based on static analysis (similarly to the relevant literature, e.g., [12]), we focus only on basic blocks, while the total application's time/energy is calculated based on a combination of these blocks. As a result, this type of operations is not taken into account in a different way and they are treated as part of the operations described in the features presented in this section. Finally, *mov* instructions are used both in memory operations and in arithmetic operations (data exchanges between registers); as a result, further classification is not needed.

Figure 3 presents the average execution time of the instruction, on two ARM processor-based embedded systems, namely the Nvidia Jetson TX1 (ARM-Cortex A-57) and the Raspberry Pi 4 (ARM-Cortex A-72). The same behavior is observed in both processors. These results highlight the reasons that led us to select the features described above. The same procedure should be followed for adding other devices in the proposed tool to determine the classification of the assembly instructions in the proper way for characterizing the execution overhead in the new platform.

Apart from the kind of the assembly instructions, also the *order* highly affects the performance. To model the instructions order and to build simple and generic features capable of being used to estimate the performance and energy for various platforms and architectures, we adopted a *sliding window* approach. More precisely, a window "slides" through the assembly instructions of the block under analysis and each combination of instructions types corresponds to a new feature, as it is depicted schematically in the illustrative example presented in Fig. 4. One may argue that, beyond the type of instruction, further information is needed, such as the registers being used in each instruction, the location of the data accessed, etc. However, we decided to keep the features simple and relatively few to support adding new devices and architectures and retraining models easily, resulting to a generic solution that offers flexibility and rapid-prototyping, perhaps with a slight effect on the quality of the results. The number of features grows exponentially depending on the window size and the different types of instructions. As mentioned before, the basic instruction categories are 3

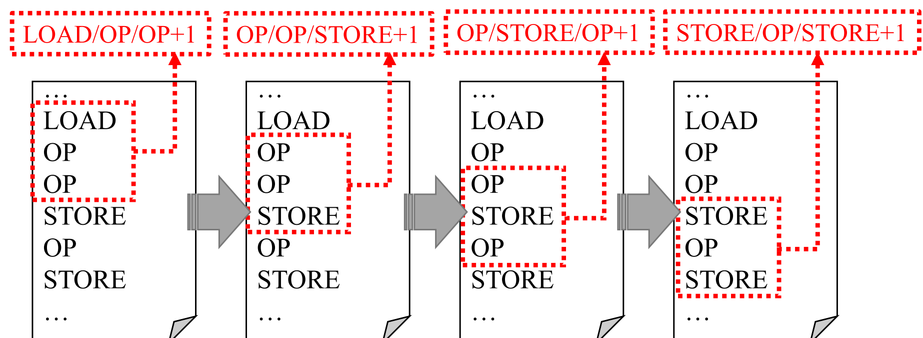

Fig. 4 Proposed sliding window method for modeling the order of the instructions

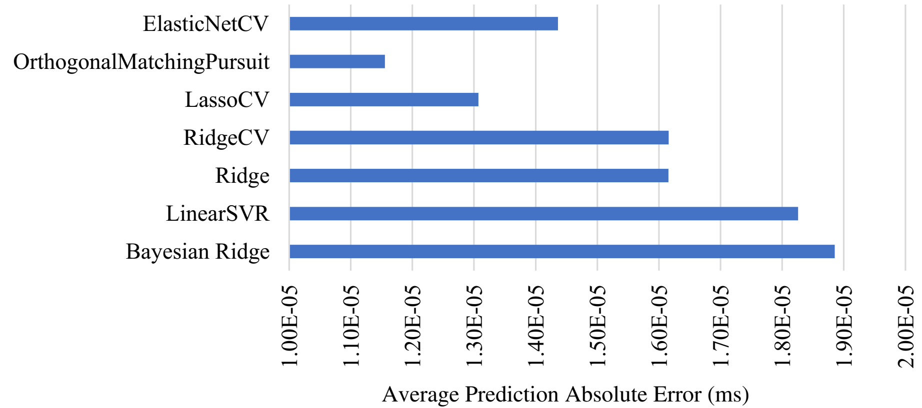

Fig. 5 Comparison of alternative models for estimating execution time

(LOAD, STORE, and OP), as the rest of them are subcategories of the OP instructions. Our final design choice is a 3-instruction-size sliding window leading up to $3^3 = 27$ new features.

In Section "Evaluation", we highlight the increase of the proposed tool efficiency as each of the features presented in this section is added in the estimation models.

Dataset

First of all, we should describe the way that the dataset is structured, which is considered crucial for developing the proposed methods of estimating energy and performance on different embedded platforms and supporting the proposed framework.

The dataset used in the present work includes synthetic randomly generated loops. We developed a python script that generates C/C++ for loops that perform single numbers, matrix, and vector operations. The number of the matrices, the data types, and the operations are random. The loops include hundreds of millions iterations to measure the execution time and the energy consumption of one iteration and consequently of the basic block as accurately as possible. All the synthetic programs are compiled using the same optimization flags (O0). The reason for this selection is the fact that the randomly generated programs usually involve calculation insides the body of the loop, that are not affected by the loop, which means that their values do not change from one iteration to another. In this case, if we use another optimization flag, the compiler will place this functionality out of the loop to avoid the meaningless overhead. As a result, this would affect our measurements, as we have large loops to repeat the execution of the entire loop body (code block) and make accurate measurements.

The final training set includes the most representative data-points from the generated dataset. This process is performed using a *k*-means clustering for dataset pre-processing, selecting one data-point from each cluster. This procedure is considered necessary as the inherent random characteristics of the generated code can lead to very similar data-points (source code blocks) which can cause model over-fitting.

In this manuscript, we focus on estimating the performance and energy of a basic block that does not require any dynamic information such as the actual number of instructions being executed, the number of iterations of a loop, etc. Regarding the cache memory behavior, which requires a dynamic analysis too, similarly to related approaches found in the literature [12], an ideal cache memory is taken into account, which means that we assume that only cache hits occur in a loop body.

Estimation Models

In this subsection, we present a comparison between alternative estimation models. All the tests are based on our generated dataset, which is splitted into training and test sets, for performing cross-validation. The measurements are performed on the Nvidia Tegra TX1 platform, which incorporates an ARM-Cortex A57 embedded processor as well as an on-board energy sensor. The final accuracy of the proposed models is also evaluated on real-life applications and well-known benchmark suites in Section "Evaluation". As mentioned in Section "Dataset", a *k*-means clustering procedure is adopted for avoiding over-fitting due to very similar data-points in the dataset. As a result, the number of the clusters is also a design choice that must be explored.

To select the regression estimation model, we compare the accuracy of different models. The accuracy of the seven most suitable models for estimating the execution time and energy consumption is depicted in Figs. 5 and 6 respectively. For each model, we present the Mean Absolute Error between the actual value that corresponds to the basic blocks from the test-set and the predicted values. According to these results, we might conclude that the *Orthogonal Matching Pursuit* model can achieve the best results compared with

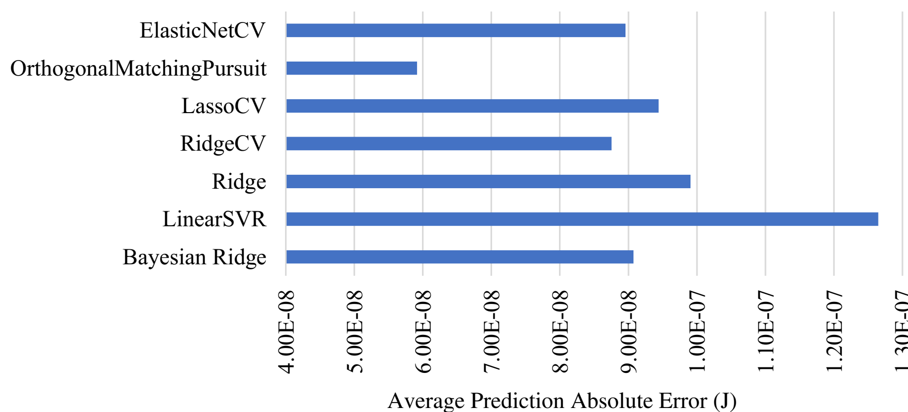

Fig. 6 Comparison of alternative models for estimating energy consumption

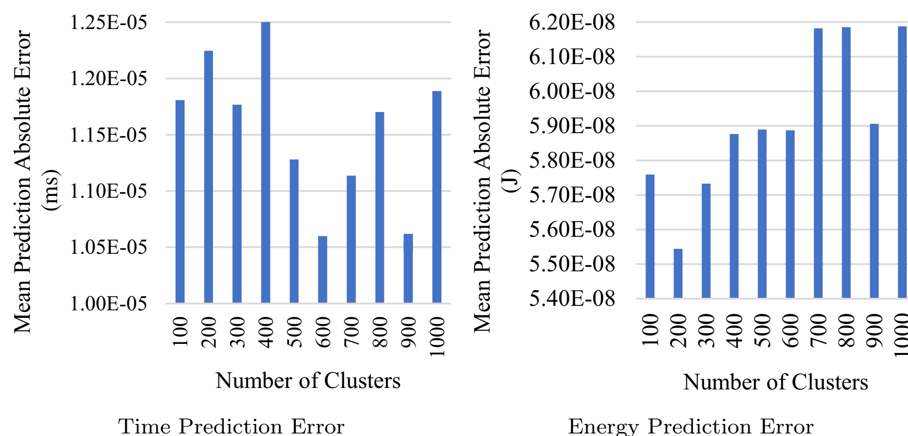

Fig. 7 Selecting the number of clusters for cleaning the dataset

the rest of the models achieving an error less than 1.2×10^{-5} ms regarding the execution time and less than 6×10^{-8} J with regard to the energy consumption.

Figure 7 depicts the impact of choosing the number of clusters that will be used in the clustering process and correspondingly the number of data-points that will be included in the final dataset. According to these results, we see that selecting 600 clusters (the 600 most representative data-points) improves the execution time prediction, while choosing 200 clusters leads to the best results in terms of energy consumption estimation.

The user of the proposed tool can add new devices on the framework by executing the random generated dataset in the new platform and by giving the results in the proposed tool. The estimation models are then re-trained and the new platform can be chosen from the list for making performance/energy estimations. Although the features are generic enough to be applicable on most platforms and different architectures, the user can make also changes in the feature list to characterize the new architecture better if needed to achieve higher accuracy. Finally, making energy consumption estimations requires an energy sensor integrated in the new platform; otherwise, only execution time estimations will be supported by the tool.

Evaluation

This section presents an experimental evaluation of the introduced framework, to highlight the efficiency of the proposed mechanism to estimate the performance and the energy consumption of applications running on embedded devices. For the purposes of the present experiments, applications (i.e., benchmarks) were retrieved from the Rodinia Suite [3]. Rodinia is a popular benchmark suite that provides a set of applications for the study of heterogeneous systems. The benchmark provides publicly available programs in multiple versions, coupled with sets of data. Each application has different inherent architectural characteristics, which affect the performance and energy consumption. Rodinia is a widely used suite especially by researchers in the field of embedded systems and it is considered as a reliable benchmark for demonstrating, testing, and presenting research results and comparisons. For this manuscript, only the CPU versions are taken into account and more specifically single-core applications.

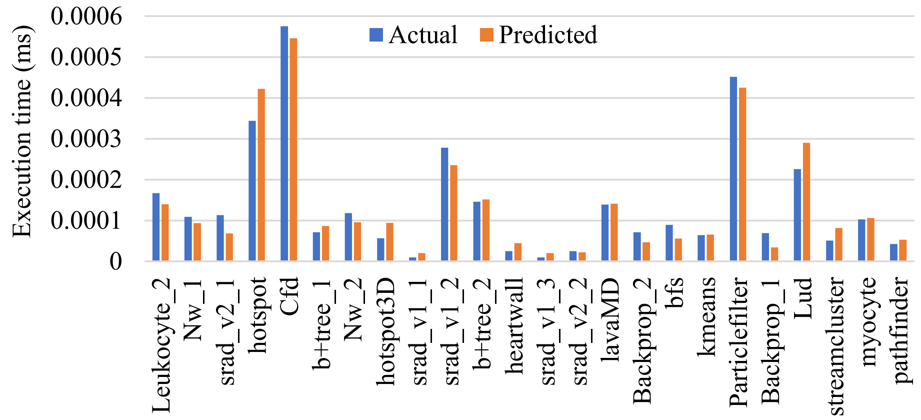

Fig. 8 Execution time estimation for most important basic blocks of Rodinia benchmark suite

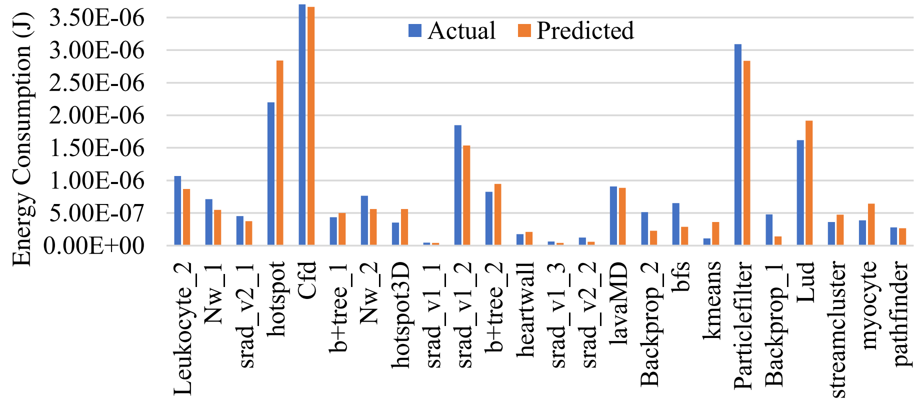

Fig. 9 Energy estimation for most important basic blocks of Rodinia benchmark suite

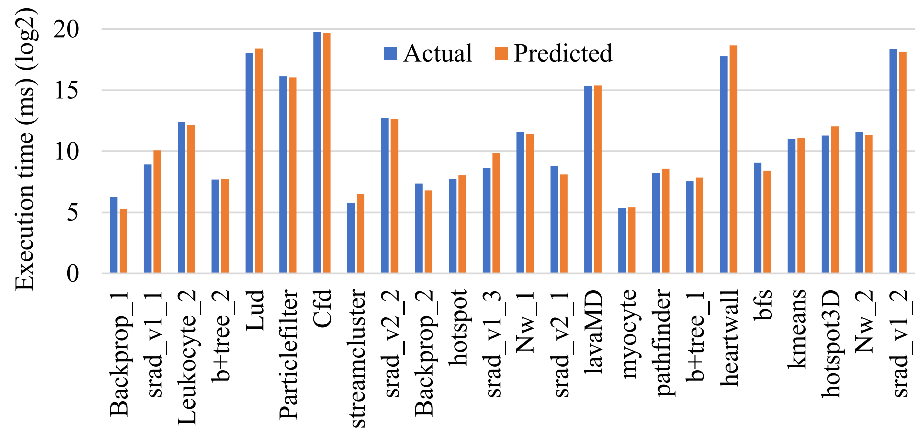

Fig. 10 Execution time estimation for most important loops of Rodinia benchmark suite

Figures 8 and 9 present the execution time and energy consumption estimation results, respectively, for running the most important basic blocks of the Rodinia applications. The code blocks selected to be analyzed are the bodies of the most computational intensive loops of the applications that have a large impact on the total program performance and energy. The applications are executed on the ARM-Cortex A57 of the Nvidia Tegra TX1 platform and the energy is calculated through the integrated energy sensor. According to these results, we might conclude that the estimation accuracy can be considered acceptable (R^2 score ≥ 0.92).

Figures 10 and 11 show the time and energy estimation (respectively) for executing the loops (after the user sets the actual number of iterations). Finally, Fig. 12 depicts the performance and energy estimation for executing the entire Rodinia applications. Based on these results, we might claim that the proposed tool achieves very

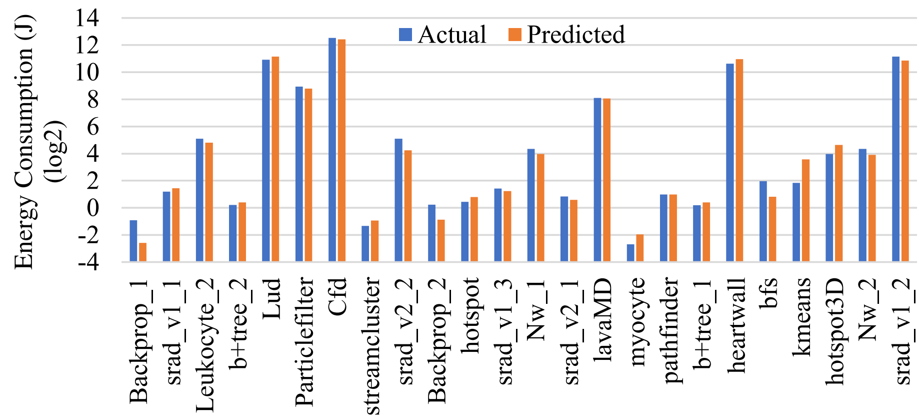

Fig. 11 Energy estimation for most important loops of Rodinia benchmark suite

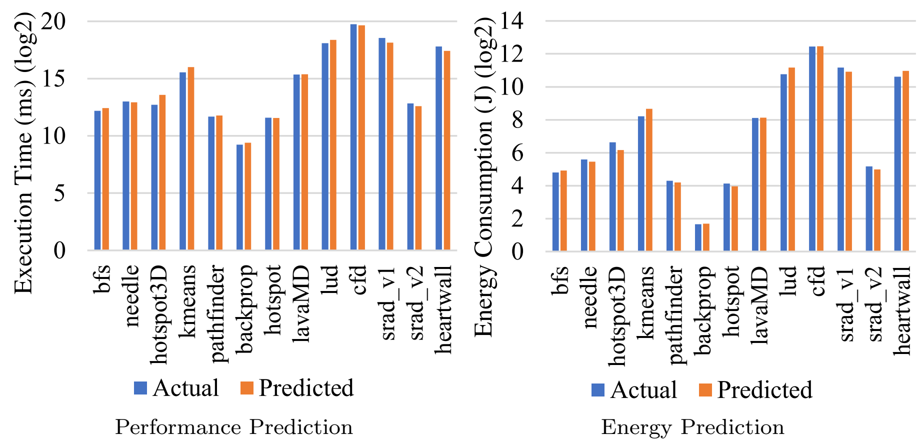

Fig. 12 Performance and energy estimation on Rodinia benchmark applications (Nvidia TX1)

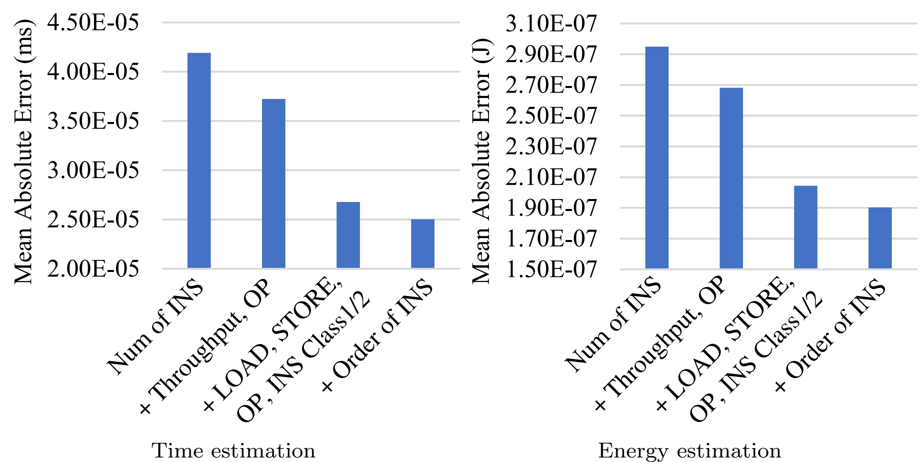

Fig. 13 Impact of the selected features on the efficiency of the tool

promising results as it can estimate the execution time and energy consumption with an acceptable accuracy, without needing to run the application. The average accuracy is 85.4% for estimating time and 80.4% for estimating energy consumption.

Figure 13 highlights the importance of each additional feature selection described in Section "Features". For the purposes of this experiment, we build the estimation models using different sets of features in the training set and we evaluate the efficiency of the tool on estimating the execution time as well as the energy consumption of the Rodinia basic blocks on Nvidia Tegra TX1 (ARM-Cortex A57). These results show how the average absolute error decreases as more features are added, each of which represent different important characteristics of the applications. In the first case, only the number of instructions is used as a feature. Then,

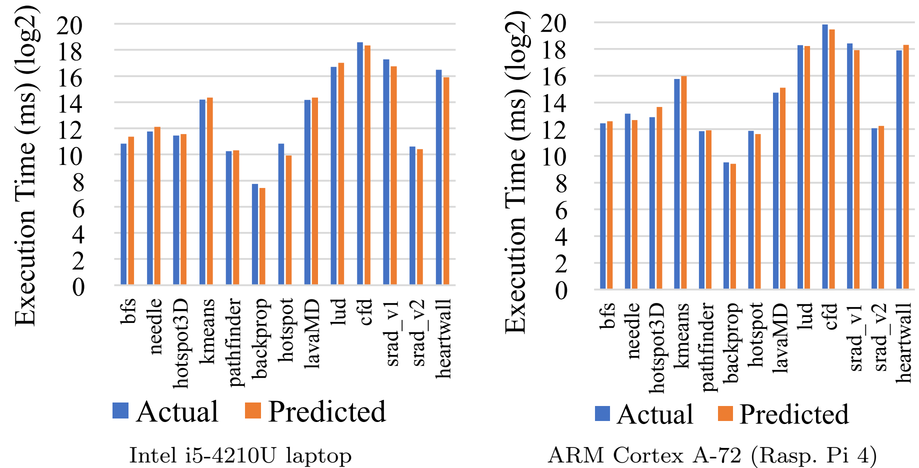

Fig. 14 Performance estimation on other CPU-based systems

we added the LLVM-mca estimated throughput as well as a separate feature for the OP operations (the rest of the instructions are loads and stores). Afterwards, we made the features more special (LOADS, STORES, OP class 1, and OP class 2), and finally, the order of the instructions is added in the way described in "Features". It should be mentioned here that the selected features have to be specific enough for having better estimation accuracy but also generic enough for supporting a large range of architectures (with different instruction sets).

Finally, we added two new CPU-based systems following the same procedure for including a new platform in the proposed tool. The first one is a well-known embedded system and more precisely the Raspberry Pi 4 that includes an ARM-Cortex A-72. For this platform, we estimated only execution time, as an energy sensor was not available. Furthermore, to evaluate the capabilities of the proposed approach to make acceptable estimations for completely different architectures, we decided to add a personal laptop with an Intel i5 4210U. The results for estimating the performance of the entire Rodinia applications are presented in Fig. 14. According to these results, we might conclude that the proposed method can support importing CPU platforms easily and make acceptable estimations. The average percentage error for estimating the time and energy for the Rodinia applications platforms is below 20% compared to the actual values.

Conclusion

In this work, a framework for supporting the estimation of performance and energy consumption of applications was introduced. The framework is based on static analysis and is easily extensible to support many CPU devices. For demonstration purposes, each sub-component of the proposed tool is evaluated on a widely used ARM-based device using a well-known benchmark suite, while the extension of the tool for supporting two more CPUs is presented. Based on our experimentation, we validate the capabilities of the proposed solution as it achieves acceptable results without the necessity of running the applications or requiring accurate prior hardware modeling.

Acknowledgements This work has received funding from the European Union's Horizon 2020 research and innovation programme under Grant agreement No. 780572 SDK4ED (https://www.sdk4ed.eu)

References

1. Bazzaz M, Salehi M, Ejlali A. An accurate instruction-level energy estimation model and tool for embedded systems. IEEE Trans Instrum Meas. 2013;62(7):1927–34.
2. Callou G, Maciel P, Tavares E, Andrade E, Nogueira B, Araujo C, Cunha P. Energy consumption and execution time estimation of embedded system applications. Microprocess Microsyst. 2011;35(4):426–40.
3. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K. Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on workload characterization (IISWC), 2009; pp. 44–54. IEEE.
4. Digkas G, Lungu M, Avgeriou P, Chatzigeorgiou A, Ampatzoglou A. How do developers fix issues and pay back technical debt in the apache ecosystem? In: 2018 IEEE 25th International Conference on Software analysis, evolution and reengineering (SANER), pp. 153–163. IEEE 2018.
5. Eder K, Gallagher J. Energy-aware software engineering. In: ICT-energy concepts for energy efficiency and sustainability, 2017; pp. 103–127 2017.
6. Ferdinand C., Heckmann R. (2004) aiT: Worst-Case execution time prediction by static program analysis. In: Jacquart R. (eds) Building the information society. IFIP International Federation for Information Processing, vol 156. Springer, Boston, MA. https://doi.org/10.1007/978-1-4020-8157-6_29.

7. Gelenbe E, Zhang Y. Performance optimization with energy packets. IEEE Syst J. 2019;13(4):3770–80.
8. Georgiou K, Xavier-de Souza S, Eder K. The iot energy challenge: a software perspective. IEEE Embed Syst Lett. 2017;10(3):53–6.
9. Georgiou S, Rizou S, Spinellis D. Software development lifecycle for energy efficiency: techniques and tools. ACM Comput Surv (CSUR). 2019;52(4):1–33.
10. Jung C, Rus S, Railing BP, Clark N, Pande S. Brainy: effective selection of data structures. ACM SIGPLAN Not. 2011;46(6):86–97.
11. Li X, Liang Y, Mitra T, Roychoudhury A. Chronos: a timing analyzer for embedded software. Sci Comput Program. 2007;69(1–3):56–67.
12. Mendis C, Renda A, Amarasinghe S, Carbin M. Ithemal: accurate, portable and fast basic block throughput estimation using deep neural networks. In: International Conference on machine learning, 2019; pp. 4505–4515.
13. Meng K, Norris B. Mira: a framework for static performance analysis. In: 2017 IEEE International Conference on cluster computing (CLUSTER), 2017; pp. 103–113. IEEE.
14. Papadopoulos L, Marantos C, Digkas G, Ampatzoglou A, Chatzigeorgiou A, Soudris D. Interrelations between software quality metrics, performance and energy consumption in embedded applications. In: Proceedings of the 21st International Workshop on software and compilers for embedded systems, 2018; pp. 62–65.
15. Papadopoulos L, Soudris D, Walulya I, Tsigas P. Customization methodology for implementation of streaming aggregation in embedded systems. J Syst Arch. 2016;66:48–60.
16. Pinto G, Castor F. Energy efficiency: a new concern for application software developers. Commun ACM. 2017;60(12):68–75.
17. Procaccianti G, Fernández H, Lago P. Empirical evaluation of two best practices for energy-efficient software development. J Syst Softw. 2016;117:185–98.
18. Siavvas M, Gelenbe E, Kehagias D, Tzovaras D. Static analysis-based approaches for secure software development. In: International ISCIS Security Workshop, 2018; pp. 142–157. Springer, Cham
19. Siavvas M, Marantos C, Papadopoulos L, Kehagias D, Soudris D, Tzovaras D. On the relationship between software security and energy consumption. In: 15th China-Europe International Symposium on software engineering education 2019.
20. Siavvas M, Tsoukalas D, Jankovic M, Kehagias D, Chatzigeorgiou A, Tzovaras D, Anicic N, Gelenbe E. An empirical evaluation of the relationship between technical debt and software security. In: 9th International Conference on Information Society and Technology (ICIST), 2019; vol. 2019.
21. Tsoukalas D, Kehagias D, Siavvas M, Chatzigeorgiou A. Technical debt forecasting: an empirical study on open-source repositories. J Syst Softw. 2020;170:110777.
22. Zheng X, John LK, Gerstlauer A. Accurate phase-level cross-platform power and performance estimation. In: Proceedings of the 53rd Annual Design Automation Conference, 2016; pp. 1–6.
23. Zheng X, Vikalo H, Song S, John LK, Gerstlauer A. Sampling-based binary-level cross-platform performance estimation. In: Design, Automation & Test in Europe Conference & Exhibition