Statement-Level Timing Estimation for Embedded System Design Using Machine Learning Techniques

Vittoriano Muttillo, Paolo Giammatteo, Vincenzo Stoico
DEWS Centre of Excellence, University of L'Aquila, L'Aquila, Italy
{vittoriano.muttillo,paolo.giammatteo}@univaq.it,vincenzo.stoico@graduate.univaq.it

ABSTRACT

During the initial design phases of an embedded system, the ability to support designers using metrics, obtained through a preliminary analysis, is of fundamental importance. Knowing which initial parameters of the embedded system (HW or SW) influence such metrics is even more important. The main characteristic of an embedded system that typically designers need to measure is the embedded SW (i.e., functions) execution time, used to describe the final system's performance (i.e., timing performance metric). The evaluation of such a metric is often a critical task, relying on several different techniques at different abstraction levels. Furthermore, in the era of Big Data, the use of Machine Learning methods can be a valid alternative to the classic methods used to evaluate or estimate metrics for temporal performance. In such a context, this paper describes a framework, based on the use of Machine Learning methods, to calculate a statement-level embedded software timing performance metric. Results are compared with those obtained with different approaches. They show that the proposed method improves the estimation accuracy for specific processor classes while also reducing estimation time.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; **Embedded hardware**.

KEYWORDS

timing performance prediction, machine learning, embedded system, feature analysis

1 INTRODUCTION

In the last thirty years there has been an exponential increase in the spread and evolution of information technology. In this regard, it is undoubtedly underlined the spiraling of Embedded systems [1]. The problem of determining an embedded system's characteristic is a task with an effort that should not be underestimated, especially during the early design phases. Indeed, working on a higher abstraction level (i.e., *Electronic System-Level, ESL*) is needed to early estimate HW/SW performance [2]. Furthermore, in the era of Big Data, the use of Machine Learning (ML) methods [3, 4] in this context can be a valid alternative to the classic methods to estimate metrics for embedded system performance [5–7]. So, according to this scenario, this work tries to answer the following research questions:

(1) RQ_1: How to extrapolate a meaningful timing performance metric for Embedded platforms without Instruction Set Simulators (ISSs) or the target device, which is time consuming?
(2) RQ_2: How to reduce estimation time and increase accuracy for embedded timing software performance predictions using Machine Learning techniques?

RQ_1 relies on the considered timing performance metric extraction [8], and the use of ISS instead of the final HW/SW embedded platforms can reduce the time needed for estimation step, depending on the simulator accuracy. RQ_2 considers to replace ISS or selected HW/SW platforms with ML algorithms to reduce design time and increase accuracy w.r.t. the traditional approaches.

Thus, this work presents an approach for timing performance estimation. The aim is to extract information about the main practice used in system-level design flows in order to reduce the time needed for the initial activities, reducing also estimation errors without an extensive and deep analysis of the final hardware/software platforms. A timing performance metric related to high-level C programming language statements will be exploited, as presented in [9]. In addition, a framework that helps to calculate this metric for a given function will be presented. Additionally, this framework can automatically generate large amounts of constrained random inputs and evaluate statistics on the metric itself. The framework exploits ISSs to evaluate several features from the source code, while static code analysis is used to enrich the dataset. The number of clock cycles needed to execute the program, the number of executed C statements with profiling on the host architecture, the Cyclomatic and Healsted number are a subset of the considered features selected for the following ML activity. These big data have been analyzed through several statistical methods, and useful results encourage exploiting this approach inside an ESL design flow.

2 PRELIMINARIES

In the context of Embedded System Design [1], some of the main challenges are related to accuracy and simulation/estimation time associated with estimation methods at different abstraction level [10]. The higher the abstraction level, the lower the accuracy and the estimation time, depending on adopted estimation models and simulators [11]. Different abstraction levels can be considered to perform timing estimation, and several approaches have been proposed in the literature. Malik et Al. [12] explored different performance metrics that need to be considered in embedded software performance analysis and examined a wide range of techniques (e.g., path analysis techniques, system utilization analysis techniques). The

ICPE '21, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450258

work in [13] presents two approaches to solve the performance estimation problem: (1) a source-based approach that uses compilation onto a virtual instruction set, and (2) an object-based approach that translates the assembly generated by the target compiler to "assembly-level". Brandolese et Al. [5] introduce a methodology for software execution time estimation. The procedure is supported by mathematical models of C statements and statistical analysis. Alterbernd et Al. [6] introduce an approach to predict the execution time of software through an early, source-level timing analysis. The estimation is done directly from the source code, without compiling and running it, using a set of virtual instructions, and defining an abstract machine able to execute the source code. Finally, [14] introduces an approach to source-level timing estimation through elementary operations. Most of these works aim to predict timing performance behaviors using time-consuming but quite accurate simulators, mathematical prediction models at the system-level, or virtual platform emulators.

In contrast, timing prediction approaches based on ML techniques are the newest methods exploiting big training data-set to make predictions without knowledge of the considered HW platform. Oyamada et Al. [15] presents a methodology for system design and performance analysis. They use an analytic approach based on neural networks for high-level software performance estimation. The analytical estimation results in errors only up to 29.75%, with an estimated time of about 17 seconds. The work in [16] extracts performances of a small set of computers. It uses this information to develop linear regression and neural network models to predict any different considered computer's performance. They can predict the performance of regarded platforms within 3.4% average error rate. Huang et Al. [17] propose the SPORE (Sparse POlynomial REgression) methodology to build prediction models of program performance using feature data collected from program execution on sample inputs, with relative error less than 7%. Finally, the paper in [18] proposes a method through performing data mining on historical data. The authors suggest performing timing prediction using three ML techniques (i.e., Naïve Bayes, Logistic Regression, and Random Forests). These works confirm the validity of ML and data mining in general and the use of these techniques in particular for software estimation. Table 1 compares the different approaches, w.r.t. ML techniques and results. Furthermore, to the best of our knowledge, very few research works exploit these techniques for embedded system timing performance estimation, mostly for the low accuracy of these techniques. Our work tries to solve this open research problem.

3 PROPOSED APPROACH

The approach proposed in this work performs statement-level execution time estimation using statistical analysis and approximate predictions, as shown in Fig. 1. The idea behind this approach is to train the ML model, using (micro-)benchmarks, so that it can predict the performance for any given input C-code function for a given target processor. Compiler and program analysis tools are exploited to extract data for this goal on different real embedded devices. The ML module uses this data to solve the timing performance prediction problem. The whole framework is divided into three macro-blocks: (a) *Input Generator Block*, where the selected functions take several inputs randomly generated inside module ①. (b) *Parallel Independent Block*, where three sub-modules can be executed independently from each other. Module ② uses Gcov program to extract dynamic information about executed C statements; module ③ performs Instruction Set Simulator (ISS) execution, with the collection of clock cycles needed to execute the benchmark function set; module ④ implements static analysis, where Frama-C [19] is used to enlarge the dataset with static function parameters useful for the ML algorithm. (c) *Machine Learning Block*, the core of the estimation method.

A module that (semi)automatically generates inputs for the benchmark functions has been created in ①. In particular, for each function they are randomly generated 1000 input data sets. Moreover, for each function, different data types have been considered (i.e. *int8, int16, int32,* and *float*) to analyze the results w.r.t. the internal architecture of the considered processor. Each input data set is stored in a header file to be included in the function at compile time. After the inputs generation phase, a procedure to count the number of executed C statements is evaluated in ②. This value is obtained by performing a profiling of the benchmark functions employing the *gcov* profiler for each generated input. To obtain the total number of executed C statements for each function, a sum of the single profiling numbers is performed. The number of clock cycles needed by the target processor technology to execute each function (for each generated input) in the benchmark (and the number of executed assembly instructions, useful for energy/power analysis) is extracted in ③. Depending on the target processor technology there is the need for an *Instruction Set Simulator* (ISS) or an *High Level Synthesis* (HLS) Simulator for Special Purpose Processors (SPP). The latter is not part of this work. The last data are evaluated using Frama-C tool and other useful script in ④, integrated inside the whole framework instance. Finally, all the generated data output artifacts are sent to ⑤ where the information is organized, selected and exploited for model training, test and validation, as described below.

3.1 Dataset Preparation and Feature Extraction

In this paper, several features have been chosen to provide a statistical analysis of collected data (i.e., distribution parameters). Several Python scripts have been created to automate all CSV file's statistical analysis, integrated into the last framework module ⑤. To guarantee unbiased data and correct ML training activities, feature analysis is applied to the output framework dataset (i.e., 33 features) that can be clustered w.r.t. the number of code line (SLOC), input data type, Halstead Complexity, Cyclomatic Complexity, functions reachability and program profiling, as described in [20]. Once the dataset is created, four regressor models are considered in order to perform the feature analysis, taking inspiration from [21]. In particular, we considered the *Random Forest Regressor*, *Extra Trees Regressor*, *Gradient Boosting Regressor* and *Ada Boost Regressor*, exploiting the python *scikit-learn* package. The average value of the previous results [21], reduced by the confidence interval value of 99%, has been used as the lower bound to select the most important features. After the feature analysis process, the most significant features will be taken into consideration. ML algorithms will be

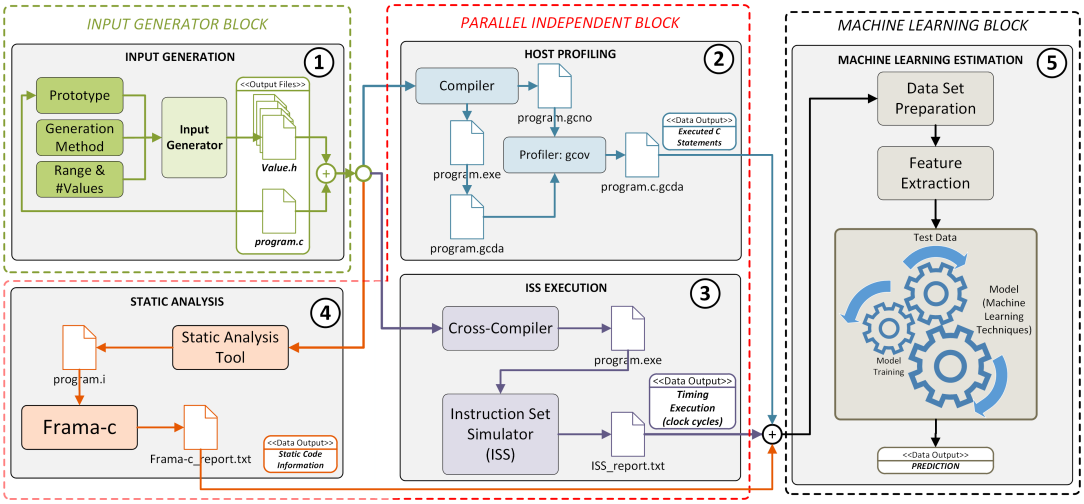


Figure 1: Proposed Approach.

Table 1: ML timing performance estimation comparison (N.A. means Not Available).

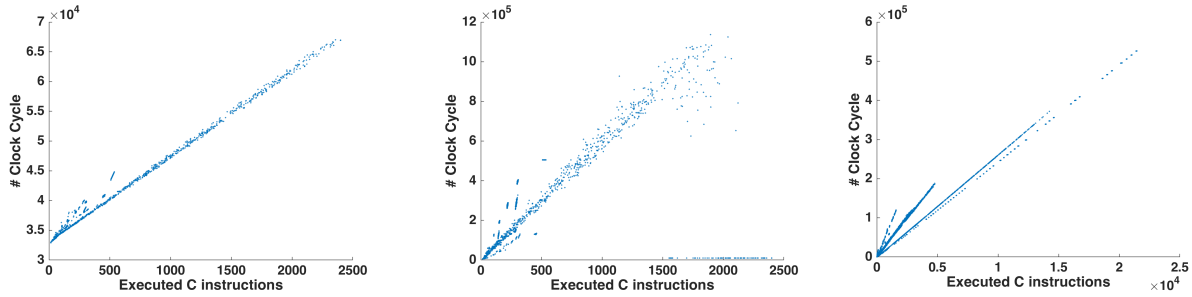
Work	Domain	Prediction Models	Data Preparation	Benchmark	Platform	Accuracy	Training Time	Estimation Time
[15]	Embedded	Neural Network	N.A.	Custom Benchmark	ARM946	up to 29.75%	Slow	up to 17s
[16]	General Purpose Computer	Linear Regression (LR) Neural Network (NN)	Clementine Software	SPEC2000	AMD Opteron Intel Pentium D and 4 Intel Xeon	LR: 1.5% ... 3.5% NN: 1.16% ... 5.94% (Best Case)	N.A.	N.A.
[17]	General Purpose Computer	Sparse Polynomial Regression	SPORE LASSO Method	Lucene Find Maxima Segmentation	Generic CPUs	up to 7%	N.A.	N.A.
[18]	General Purpose Computer	Naïve Bayes (NB) Logistic Regression (LR) Random Forest (RF)	Wrapper Feature Selection	NASA software Benchmark	Generic CPUs	NB: 17% LR: 19% RF: 14%	N.A.	N.A.
This Work	Embedded	BAT, BOT, FT, LR, SVM	Average Score Value	Custom Benchmark	8051 LEON3 ATmega328/P	up to 2.58% (Best Case)	< 10s	« 1ms

trained to predict the timing performance metric, or better the *Effective Clock Cycles* feature.

3.2 Machine Learning Techniques

The ML techniques considered, among the most used in literature [22, 23], range from regression trees to SVM, and linear regressors, with the aim to identify which of these can better predict the timing performance metric. At this stage of the work, the Matlab app Regressor Learner [24] has been used. In particular, we considered five ML methodology, according to those available: Linear Regression (LR), Fine Trees (FT), Boosted Trees (BOT), Bagged Trees (BAT), and Support Vector Machine (SVM). We excluded Neural Network Algorithm's usage since we plan to dedicate a specific work in future time so as not to burden the discussion in this paper [25]. Furthermore, this analysis involves the usage of a different Matlab package.

Linear regression models [26] have predictors linear in the model parameters and are easy to interpret and fast for making predictions. However, the highly constrained form of these models means that they often have low predictive accuracy compared to the others. Regression Learner App uses the *fitlm* function to train *Linear* model option. Regression trees [26] are easy to interpret, fast for fitting and prediction, and low in memory usage. The Matlab App Regression Learner gives the possibility to choose among different kinds of regression trees. In this work, the *Fine Tree* option was selected, which means high flexibility, with many small leaves for a highly flexible response function (the minimum leaf size is 4). The Regression Learner App uses the *fitrtree* function to train regression trees, with the parameter *Minimum leaf size* equal to 4 and no splitting criteria for surrogate nodes. Ensemble Boosted Tree model combine results from many weak learners into one high-quality ensemble model. The approach involves a least-squares boosting methodology with regression tree learners [26]. Regression Learner App uses the *fitrensemble* function to train ensemble models and gives the possibility to set three different parameters: the *Minimum leaf size*, the *Number of learners* and the *Learning rate*, which are respectively 8, 30 and 0.1. Another kind of ensemble model with regression tree learners is the Bagged Trees [26]. Regression Learner App uses the *fitrensemble* function to train ensemble models. Here the parameter to set are two: the *Minimum leaf size* and the *Number of learners*, which are respectively 8 and 30. Support vector machines are supervised learning methods used both for classification



(a) LEON3 Corr. Scatter Plot. (b) 8051 Corr. Scatter Plot. (c) ATmega328/P Corr. Scatter Plot.

Figure 2: Correlation plot w.r.t. different processors and float data type

and regression [26]. Among the advantages of the support vector machines approach, it is effective in high dimensional spaces and in cases where the number of dimensions is greater than the number of samples. Regression Learner App uses the *fitrsvm* function to train SVM regression models and, in this case, the parameter *Kernel function* is set to *Linear*.

4 EXPERIMENTAL RESULTS

The proposed approach has been evaluated first in the SW domain by considering General Purpose Processor (GPP) technology. In this work, three GPPs, and their related ISSs, have been analyzed: ***LEON3***, a 32-bit synthesizable RISC soft-microprocessor with a Harvard architecture, and a simulated system clock of 75 MHz. Cobham Gaisler offers TSIM System Emulator as an accurate ISS of LEON3 processors, with gcc compiler (in this work, the default optimization flag $-O0$ has been used); ***Intel 8051 micro-controller***, an 8-bit micro-controller with MCS-51 CISC instruction set and Harvard Architecture; The Dalton Instruction Set Simulator (ISS) [27] has been used to simulate programs written for the 8051 and to collect statistics, with SDCC (Small Device C Compiler) compiler; ***picoPower CMOS 8-bit AVR ATmega328/P***, with an 8-bit RISC-based processor core and Harward architecture. The SimulAVR [28] program has been used as a simulator for the Atmel AVR family of microcontrollers, with gcc compiler (in this work, the default optimization flag $-O0$ has been used).

The benchmark execution has been done with a microprocessor software simulation using each processor's ISS, as presented above.

4.1 Dataset Preparation and Feature Extraction Results

A benchmark composed of control-dominated and data-dominated applications was used to train and test the estimator [15] and finally validate the approach. The benchmark is composed of 15 different algorithms taken from [29] for training and test, with a total amount of $6 * 10^4$ samples, and 12 different algorithms taken from [30] for validation, with a total amount of $48 * 10^3$ samples. Table 2 describes the functions used in the experiments. For each function, different data types have been considered (*int8, int16, int32,* and *float*). Indeed, timing performance metric changes w.r.t. the dimension of data [11].

Table 2: Functions Set.

Sort and Search	binarysearch, bubblesort, insertionsort, mergesort, quicksort, selectionsort
Numerical	matrixmul, gcd, bankeralgorithm
Networking	astar, bellmanford, bfsdfs, djikstra, floydwarshall, kruskal
Data Processing	allpass, bitrev, can, conv, dir Viterbialgorithm, tap, wave, wrap

Fig. 2 shows the scatter plot related to the Pearson correlation for different processors. The three figures show a strict correlation between C statements and clock cycles. Regarding the other points (the ones under the main linear regression line), the deviation depends on function implementations that introduce different behaviors compared to the significant distribution. These points introduce errors inside the timing estimation activity and are related to the internal processor micro-architecture (i.e., 8/16/32 bit architecture, pipeline, number of registers, etc.).

Table 3 shows the Pearson correlation and slope values between clock cycles and executed C statements. The high correlation values reported in Table 3 between the Clock Cycles and the number of C statements prompts us to explore further the processor characteristics and to exploit ML algorithms for timing performance estimation.

Table 3: Correlation and Slope Analysis results (p-value $\ll$ 0.001 for every processor, so that we can reject the null hypothesis and the statistical analysis is highly significant).

Function	int8	int16	int32	float	Tot.
LEON3 Corr.	0.9939	0.9872	0.9277	0.7465	0.9631
MPU8051 Corr.	0.9939	0.9871	0.9276	0.7465	0.9631
ATmega328/P Corr.	0.7465	0.9939	0.9871	0.9277	0.9631
LEON3 Slope	10.8838	9.4241	10.9702	15.1550	88.9492
MPU8051 Slope	88.2752	110.5197	183.8507	389.2319	88.9492
ATmega328/P Slope	24.7353	11.197	18.1595	14.0614	88.9492

In Fig. 3 is reported the result of the feature analysis. The figure (green line) shows the arithmetic mean behavior between the four algorithms results [21]. Some features show prominent behavior

compared to others. To select the most important, a selection criterion is introduced, as described in Section 3.1. From Fig. 3 it is worth noting that two feature dominates the other ones. These two features are related to the dynamic execution of the code. To introduce features connected to static code analysis, we decided to remove the "Executed Assembly Instructions" and "Executed C instructions" features and to apply the selection criteria to the remaining features again. The solid red line represents the average value calculated without the two dominant features. Since some features are close to the solid red line in Fig. 3, we included features with a mean score value holds within the confidence interval of 99% (i.e., the dashed lines), eliminating all those features with a value less than the lower confidence interval value.

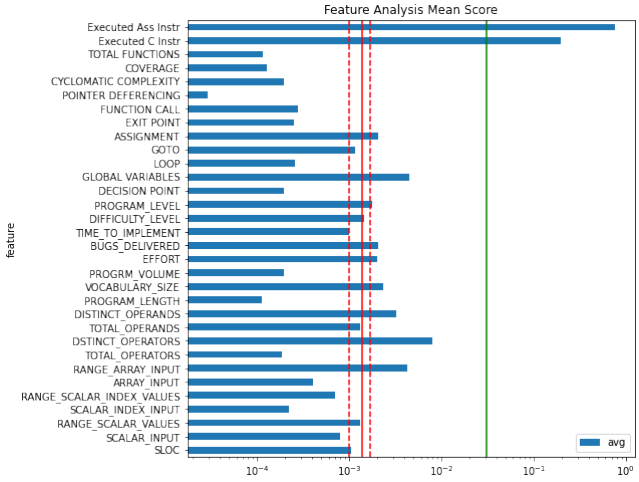


Figure 3: Feature analysis (X-axes represent the percentage score from 0 to 1).

Respect to each regressor algorithms the variance is 0.0210 for *Random Forest Regressor*, 0.0169 for *Extra Trees Regressor*, 0.0202 for *Gradient Boosting Regressor* and 0.0188 for *Ada Boost Regressor*. The most prominent value is for *Random Forest Regressor*, while the feature *GLOBAL VARIABLES* has the larger variance than others, considering all the regressor.

4.2 Machine Learning Training and Test Results

Once the less significant features were eliminated from the dataset for each processor, the ML algorithms listed in the previous paragraph were trained to define a model capable of predicting the output performance variable, i.e., the Clock Cycles. The dataset generated from the 15 functions inside the training and test benchmark is divided into 80% for training and 20% for testing for each processor. After the training process, in Fig. 4 are reported the prediction values Vs. the real values for a subset of devices and ML models trained. The good agreement between prediction and real values for *FT* in each device is evident. Also, the *LR* and *SVM* behave well, at least for 8051 and LEON3. The *BAT*, on the other hand, provides bad results for each device. The ATmega328/P device badly behaves for every predictor except for the FT.

Testing is performed on 20% of the dataset. Fig. 5 shows the results of the Root Mean Square Percentage Error (RMSPE) for a subset of the processors. In each plot, it is possible to distinguish each ML predictor model's behavior for various types of data aggregation: by function, data type, total dataset test (All). As mentioned for the prediction vs. real values plots, the *Fine Tree* is the ML model that performs best since its RMSPE values are globally lower than all other ML models. It is worth noting that in the LEON3 and 8051 scenarios, the predictor's RMSPE is always under the 10%, while the FT and BAT for the same devices are generally under the 1%. There is just an exception w.r.t. 8051, where the int8 scenario is badly driven by bubblesort function (maybe related to some dataset noise). The ATmega328/P results are the worst scenario because the error is always $\geq 10^0$ and $\leq 10^1$ for BAT and FT, while is $\geq 10^1$ for BOT, LR, and SVM. Concerning the RMSE, the error variance ranges from 10^2 and 10^4. For the ATmega328/P, the error range is shifted to 10^3 and 10^5.

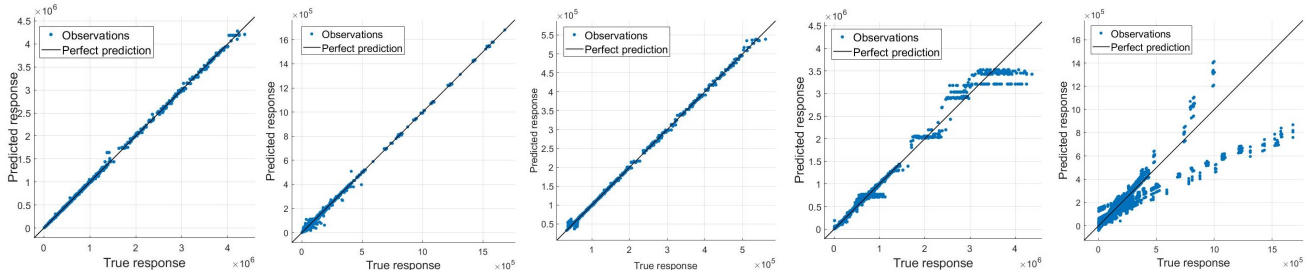
Overall, from the following analysis, it can be stated that the *SVM* does not fit well the problem and owns the worst overall RMSPE, probably since there is a lot of data w.r.t. the feature. *Boosted Trees* and *Bagged Trees* can be considered acceptable. Still, they need an optimization of the model's input parameters, especially for the BAT, if we look at Fig. 4. The *Linear Model* is good for prediction but swinging for the RMSE, as expected. The best ML model is the *Fine Tree*, which performs best for predictions and RMSPE. Also, the latter can be improved by further modifying the input parameters of the model.

As a final analysis, in Fig. 6 are reported the average prediction time considering 8051 processor (Figure a) and the average training time for each processor of the five ML models (Figure b). The average prediction time behaviour is equal for each selected processor. The best performance belongs to the *Bagged Trees*, while the *SVM* persist with worse execution timing behavior. Knowing each model's execution times is essential to understand how quickly the model is in its training and prediction phase, considering the ESL scenario where it is necessary to execute the model predictions several times. Regarding the training time, BAT and BOT are the quicker ones, under 5 *s*, LR and FT approximately 10 *s*, while the SVM takes the slowest execution time, more than 2 *h*.

4.3 Validation

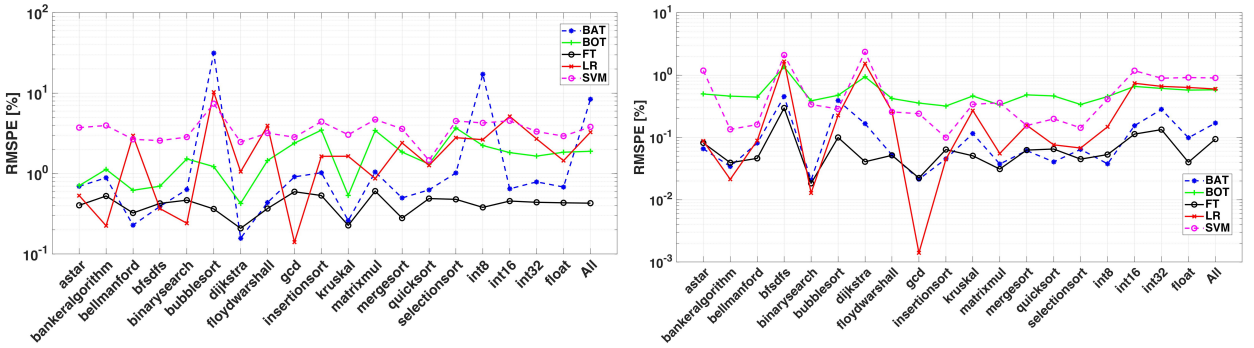
In this paragraph, the validation of the five ML models is presented. Fig. 7 report the RMSPE for 8051 and ATmega328/P. Still, it is possible to see a good behavior of the decision tree (FT) and ensemble models (BAT and BOT), unlike the *Linear* and *SVM*, which have extremely high percentage values. Regarding FT, BOT, and BAT, it is possible to notice that FT is the best considering the different processors. For the 8051, all the three best models are under the 10% of RMSPE. FT is under the 3% (with values below 1%), BOT is under 7% (with an error range greater than the FT scenario), while BAT is between 1% and 10%. For the LEON3, FT, BOT, and BAT are similar, varying between 9% and 70%. Finally, the unique model that is always under 60% is the FT for the ATmeag328/P, ranging from 1% to 60%, with an average error of about 20%.

The average time prediction analysis shows the same behavior w.r.t. the original training dataset, placing the *BAT* as the best and



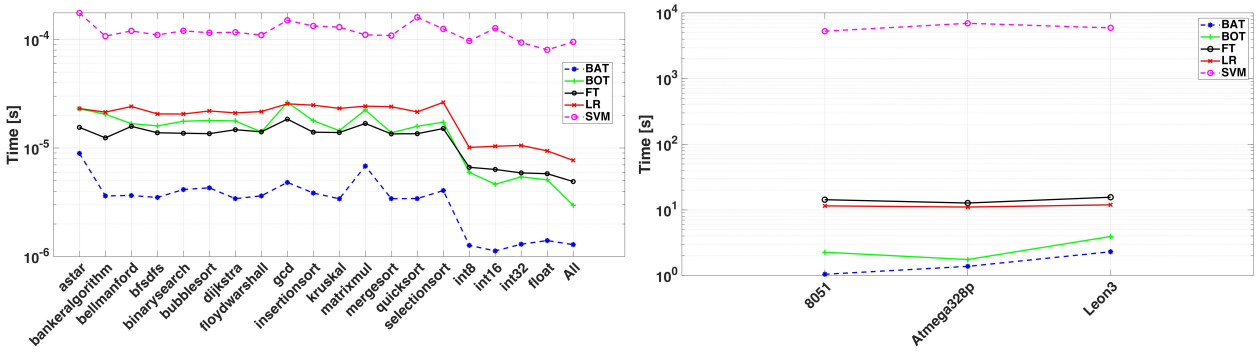
(a) 8051 FT. (b) ATmega328/P FT. (c) LEON3 FT. (d) 8051 BAT. (e) ATmega328/P SVM.

Figure 4: Predictions Vs Real Values w.r.t. different prediction models and processors



(a) 8051 RMSPE (%). (b) LEON3 RMSPE (%).

Figure 5: Test Errors w.r.t. different processors.



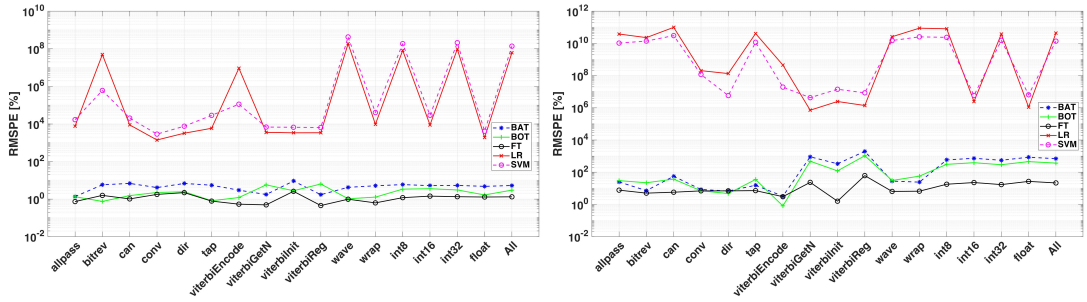
(a) 8051 Prediction Time. (b) Training Time.

Figure 6: Training and Prediction Time w.r.t. different processor technologies.

SVM the worst from a timing performance point of view. The prediction times for the validation dataset are similar to those considered in previous paragraphs, while variance is in terms of $ms - \mu s$, as shown in Fig. 8.

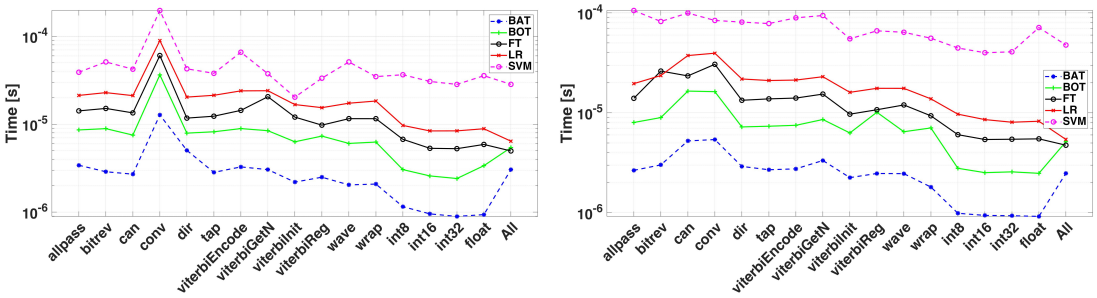
4.4 Discussion

Answer to RQ_1: The considered "meaningful" timing performance metric is the embedded software execution time (i.e., Clock Cycles) [11]. It is possible to use the approach presented in this paper to predict this metric. The idea is to use ML techniques instead of simulators/emulators. This idea is reasonable because we have noted a quite high observed correlation among the Clock Cycles, C Statements, and Assembly Instructions Executed, as shown in Fig. 2 and Table 3. Despite the high correlation values shown in the table, LEON3 and ATmega328/P present lower correlation values, respectively, to float and int8, which can lead to wrong predictions. For 8051, the lowest correlation value of float data type depends on internal microcontroller architecture (i.e., due to the absence of the Floating-Point Unit). The use of the proposed ML approach leads to feature selection problems (i.e., to find the most essential features



(a) 8051 RMSPE (%). (b) ATmega328/P RMSPE (%).

Figure 7: Validation Errors w.r.t. different processors.



(a) 8051 Prediction Time. (b) LEON3 Prediction Time.

Figure 8: Prediction Time w.r.t. different processor technologies and validation data set.

considering only the software code). From Fig. 3, we can state that the most important features that contribute to the prediction model are the Executed Ass. Instr. (Assembly Instructions) and Executed C Instr. (C Instructions). The only Cyclomatic Complexity parameters considered are ASSIGNMENT and GLOBAL VARIABLES, which depends on the input data type. The Halsted Complexity parameters, the Input Data Type, and SLOC are included almost totally, while the Function Reachability parameters (COVERAGE and TOTAL FUNCTIONS) do not reach the considered threshold. The minimum score feature is the POINTER DEFERENCING since there are not many dereferencing pointer operators inside the considered benchmark.

Answer to RQ_2: It is necessary to exploit several ML algorithms and analyze them to reduce estimation time and increase accuracy using ML. Regarding the Fig. 4, we can notice that the models based on the regression trees approach behaves better than the other ones. The same behaviour is also present in the plots reported in Fig. 5 (training and test) and Fig. 7 (validation). 8051 shows the best prediction behavior in terms of RMSPE. For training and test FT, BOT and BAT are under 1%, while validation of the 8051 timing performance prediction errors ranges from 1% to 10%. These low errors depend on 8051 microcontroller architecture (i.e., 8 bit), instructions set (i.e., MCS-51 CISC), and limited register size (i.e., 8 bit). For ATmega328/P, despite the architecture and register size being the same as 8051, the different instructions set (i.e., RISC) leads us to have high errors because of the reduced number of micro-code instructions that deal with a vast amount of assembly line after the compilation step. LEON3 presents good results w.r.t. training and test (i.e., all the prediction models have errors lower than 1%), but worst for validation (i.e., higher than 60%), compared to ATmega328/P (i.e., errors lower than 1% for the FT scenario). This error depends on LEON3's more complex processor architecture (i.e., 32 bit with 7 stage pipeline). Fig. 6 and Fig. 8 present the training, test, and validation prediction time, where all the results are aligned in terms of time granularity. The fast prediction model is the BOT, followed by BAT and FT, and it shows that these three models are the best in estimation and training time, and they can be improved in future works. This approach seems to be good enough to answer to RQ_2 as presented in [15]. Concerning the results in [15], Table 4 present a comparison between the two different approaches (Neural Network for [15], FT for our work). The FT estimation error is lower than the other test scenario approach, while LEON3 and ATmega328/P behave worst in the validation scenario. Also the estimation time is slower than [15], as shown in Fig. 6. Future works will continue to refine this strategy working on ML parameters and algorithms.

5 CONCLUSION

This work presented an ML-based approach to predict the timing performance of embedded processors. The modular framework helps the designer extracting static and dynamic code information, used by the next ML module to guess selected embedded processors' timing performance. The results show that the proposed approach

Table 4: Estimation Error Comparison.

Processor	Max Error	Mean Error	Std Deviation
ARM946 Test [15]	29.75%	9.05%	8.90%
8051 Test	0.61%	0.42%	0.12%
ATmega328/P Test	6.03%	2.24%	1.48%
LEON3 Test	0.29%	0.07%	0.06%
ARM946 Validation [15]	N.A.	N.A.	N.A.
8051 Validation	2.58%	1.19%	0.69%
ATmega328/P Validation	65.39%	12.58%	17.79%
LEON3 Validation	73.74%	52.02%	21.36%

is good enough to predict timing behaviors for a CISC microcontroller. Simultaneously, the use of this method for more complex processors introduces estimation errors that can be reduced with more advanced ML techniques. Future works will investigate (1) the possibility to increase the total number of features introducing metrics and features able to characterize not only the input data types but also the input specific values (2) the model prediction parameters optimization (i.e., minimum leaf size and the number of learners for ensembles of trees and regression trees, kernel function for SVM, etc.); (3) overfitting problems (e.g., the LEON3 scenario) and the use of cross-validation to solve them, (4) the possibility to analyze several regression models, different from the ones considered in this work, also considering the use of Neural Networks as a concrete and useful ML algorithm alternative, (5) the usage of the proposed approach to characterize the impact of SW-monitoring systems on given processors [31], (6) the possibility to apply this approach to other domains (i.e., Cyber-Physical Systems, SW Computer performance, etc.) [32].

ACKNOWLEDGMENTS

This work has been partially supported by the ECSEL RIA 2017-783162 FitOptivis and the ECSEL-JU 2018-826610 COMP4DRONES projects.

REFERENCES

[1] Yuriy Zacchia Lun, Alessandro D'Innocenzo, Francesco Smarra, Ivano Malavolta, and Maria Domenica Di Benedetto. State of the art of cyber-physical systems security: An automatic control perspective. *Journal of Systems and Software*, 149:174 – 216, 2019.

[2] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[3] Francesco Smarra, Giovanni Domenico Di Girolamo, Vittorio De Iuliis, Achin Jain, Rahul Mangharam, and Alessandro D'Innocenzo. Data-driven switching modeling for mpc using regression trees and random forests. *Nonlinear Analysis: Hybrid Systems*, 36:100882, 2020.

[4] A. Lojo, L. Rubio, J. M. Ruano, T. Di Mascio, L. Pomante, E. Ferrari, I. G. Vega, F. K. Gürkaynak, M. L. Esnaola, V. Orani, and J. Abella. The ecsel fractal project: A cognitive fractal and secure edge based on a unique open-safe-reliable-low power hardware platform. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 393–400, 2020.

[5] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES '01)*, page 98–103, New York, NY, USA, 2001. ACM.

[6] Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Syst.*, 52(6):731–760, Nov. 2016.

[7] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4), March 2014.

[8] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, USA, 2000.

[9] V. Muttillo, G. Valente, L. Pomante, V. Stoico, F. D'Antonio, and F. Salice. Cc4cs: An off-the-shelf unifying statement-level performance metric for hw/sw technologies. In *Companion of the 2018 ACM/SPEC International Conference*, New York, NY, USA, 2018. ACM.

[10] G. Valente, T. Di Mascio, G. D'Andrea, and L. Pomante. Dynamic partial reconfiguration profitability for real-time systems. *IEEE Embedded Systems Letters*, pages 1–1, 2020.

[11] David J. Lilja. Measuring computer performance: A practitioner's guide. *SIAM Review*, 43:383–384, 01 2001.

[12] S. Malik, M. Martonosi, and Y. S. Li. Static timing analysis of embedded software. *Proceedings of the 34th Design Automation Conference*, pages 147–152, 1997.

[13] Jwahar R. Bammi, Wido Kruijtzer, Luciano Lavagno, Edwin Harcourt, and Mihai T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign. CODES 2000*, CODES '00, page 82–86, New York, NY, USA, 2000. Association for Computing Machinery.

[14] Nikolina Frid, Danko Ivošević, and Vlado Sruk. Elementary operations: a novel concept for source-level timing estimation. *Automatika*, 60(1):91–104, 2019.

[15] Marcio Oyamada, Flávio Wagner, Marius Bonaciu, Wander Cesário, and Ahmed Jerraya. Software performance estimation in mpsoc design. In *Asia and South Pacific Design Automation Conference*, volume 0, pages 38–43, 01 2007.

[16] B. Ozisikyilmaz, G. Memik, and A. Choudhary. Machine learning models to predict performance of computer system design alternatives. In *2008 37th Int. Conf. on Parallel Processing*, pages 495–502, 2008.

[17] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 1*, pages 883–891. Curran Associates Inc., 2010.

[18] Monika and O. P. Sangwan. Predicting software effort estimation using machine learning techniques. In *7th International Conference on Cloud Computing, Data Science Engineering - Confluence*, pages 92–98, 2017.

[19] *Frama-C Software Analyzers*, 2020 (accessed: 15.03.2020). https://frama-c.com/.

[20] *CC4CS-ML Open-SOurce Git Repository*, 2020 (accessed: 15.03.2020). https://github.com/vnzstc/cc4cs.

[21] I. Letteri, G. Della Penna, and P. Caianiello. Feature selection strategies for http botnet traffic detection. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 202–210, 2019.

[22] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[23] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[24] *Regression Learner App*, 2020 (accessed: 15.03.2020). https://it.mathworks.com/help/stats/regression-learner-app.html.

[25] P. Giammatteo, F. V. Fiordigigli, L. Pomante, T. Di Mascio, and F. Caruso. Age gender classifier for edge computing. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2019.

[26] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[27] *Dalton Project: 8051 microcontroller, University of California*, 2020 (accessed: 15.03.2020). http://www.ann.ece.ufl.edu/i8051/.

[28] Martin Becker, Ravindra Metta, Rentala Venkatesh, and Samarjt Chakraborty. Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors. *International Journal on Software Tools for Technology Transfer*, 02 2018.

[29] Vittoriano Muttillo, Paolo Giammatteo, Vincenzo Stoico, and Luigi Pomante. An early-stage statement-level metric for energy characterization of embedded processors. *Microprocessors and Microsystems*, 77:103200, 2020.

[30] Donatella Sciuto, Fabio Salice, Luigi Pomante, and William Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, page 55–60, New York, NY, USA, 2002. ACM.

[31] G. Valente, T. Di Mascio, L. Pomante, and V. Stoico. An esl methodology for hw/sw co-design of monitorable embedded systems: the "design for monitorability" project - work-in-progress. In *2020 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 40–42, 2020.

[32] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.