A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning

Thomas Huybrechts
University of Antwerp – imec, IDLab – Faculty of Applied Engineering, Belgium
thomas.huybrechts@uantwerpen.be

Siegfried Mercelis
University of Antwerp – imec, IDLab – Faculty of Applied Engineering, Belgium
siegfried.mercelis@uantwerpen.be

Peter Hellinckx
University of Antwerp – imec, IDLab – Faculty of Applied Engineering, Belgium
peter.hellinckx@uantwerpen.be

Abstract

The notion of the Worst-Case Execution Time (WCET) allows system engineers to create safe real-time systems. This value is used to schedule all software tasks before their deadlines. Failing these deadlines will cause catastrophic events, e.g. vehicle crashes, failing to detect dangerous anomalies, etc. Different analysis methodologies exist to determine the WCET. However, these methods do not provide early insight in the WCET during development. Therefore, pessimistic assumptions are made by system designers resulting in more expensive, overqualified hardware.

In this paper, an extension on the hybrid methodology is proposed which implements a predictor model using Machine Learning (ML). This new approach estimates the WCET on smaller entities of the code, so-called *hybrid blocks*, based on software and hardware features. As a result, the ML-based hybrid analysis provides insight of the WCET early-on in the development process and refines its estimate when more detailed features are available. In order to facilitate the extraction of code-related features, a new tool for the COBRA framework is proposed.

This paper proves the potential of the ML-based hybrid approach by conducting multiple experiments based on the TACLeBench on a first prototype. A set of annotated code features were used to train and validate eight different regression models. The results already show promising estimates without tuning any hyperparameters, proving the potential of the methodology.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Worst-Case Execution Time, Machine Learning, Hybrid Analysis, Feature Selection, COde Behaviour fRamework

Digital Object Identifier 10.4230/OASIcs.WCET.2018.5

1 Introduction

In the last decade, embedded systems have taken a more prominent role in our environment. The possible applications with these systems are endless, e.g. smart mobile devices, cars, avionics, etc. For instance, the number of devices connected in the Internet of Things (IoT) is expected to rise to 20 billion units in 2020 [19]. Unlike general purpose computers, cyber-physical system (CPS) and IoT applications require specific context related constraints on the controller units, such as energy consumption, size, real-time behaviour (i.e. execution time), etc. This guarantees affordable, reliable and safe systems. However, these requirements also have consequences on the code running on these devices.

An (autonomous) car is a perfect example of a CPS with hard real-time constraints. It is from the uppermost importance that these systems not only have correct behaviour, but also are responsive. For example, the Electronic Control Unit (ECU) of the braking system should respond to the breaking pedal before a strict deadline in order to prevent catastrophic consequences. Therefore, the Worst-Case Execution Time (WCET) of code is an important value for real-time systems. However, determining this value can not always been taken for granted, as different optimisation techniques used in embedded systems and compilers influence the deterministic behaviour of the software, such as pipelining, branch prediction, pre-emption, parallelisation, etc. As a result, the WCET analysis becomes complexer to perform. In the state of practise, these influences are often simplified or neglected and compensated with a safety margin resulting in less tight or underestimated upper bound [14].

In order to determine the schedulability of software tasks, a timing analysis is required to calculate the WCET. For instance, this value is required by the scheduler of operating systems and hypervisors to schedule all tasks within specified time frames on the system [4].

In the state of the art, there are three main WCET analysis methodologies, namely the static, measurement-based and hybrid approach [13] [18]. However, a big trade-off between accuracy and computational complexity needs to be made. We believe that the hybrid methodology is the best solution as it provides the possibility to set a balance between accuracy and computational complexity depending on the needs of the user. The implementation of this hybrid approach is integrated in our COBRA framework [12] which allows us to perform code behaviour analysis on different embedded platforms.

Nevertheless, it is difficult to acquire early insight of the WCET during development with the hybrid methodology as it relies on the physical hardware and binary code to measure the execution time. In addition, the measurement process itself is time consuming. Therefore, we want to extend the hybrid methodology by applying machine learning (ML) techniques to predict the WCET of the code blocks instead of physically measuring it on the device.

In this paper, we will firstly discuss the hybrid methodology for WCET analysis. Secondly, we present a new extended approach combining the hybrid analysis with machine learning to predict the WCET without the need to actually performing physical measurements on the device unlike the measurement-based layer. Finally, we conclude with an early stage experiment proving the potential of our approach.

2 Hybrid Methodology

The hybrid WCET analysis combines the strengths of two commonly used methodologies. On the one hand, we have the *static analysis*. This approach determines the WCET based on models of the application and the target hardware without actually executing the code itself on the platform. However, the computational complexity of creating and calculating these models rises tremendously with the size of the code base and hardware optimisations on the target platform. These models, which are not always publicly available, are required to obtain sound results with a small upper bound [20]. Therefore, the static analysis becomes infeasible as the complexity of the system increases.

On the other hand, the *measurement-based analysis* is computational less complex compared to the *static analysis*. The execution time of the program is measured by running the code multiple times with different input sets, resulting in a timing distribution. This distribution leads to three important boundaries: best-case (BCET), average-case (ACET) and worst-case execution time (WCET). Depending on the analysis cost and effort that can be afforded, the number of measurements is decided. By measuring an arbitrary limited

number of input cases, it is never guaranteed that the real WCET is detected! The accuracy will drop dramatically when the amount of measurements decreases as not all systems states are covered by the given input sets. A safety margin needs therefore to be taken into account to minimise the risk of underestimating the upper bound [20].

In order to tackle the shortcomings of the previous mentioned techniques, we are using the *hybrid methodology* which combines both approaches to estimate the WCET of a software task [3] [8] [12]. The goal is to find a balance between the computational complexity of the static layer and the accuracy issue of the measurement-based layer. To apply this methodology, the source code is split into a set of smaller entities which are called "*hybrid blocks*". Each block resembles a trace of consecutive instructions which has exact one entry and one exit point [11]. The blocks are similar to the regular "*basic blocks*" [13]. However, the size of these blocks can vary from a single instruction up to entire functions or programs depending on the accuracy and complexity we want to achieve [11]. The process starts by performing timing measurements on each block. In the second stage, all results are statically combined to acquire an estimate of the WCET.

The hybrid analysis is integrated in the *Code Behaviour fRAmework* (COBRA) tool. This open source tool is developed by the IDLab research group to examine the performance and behaviour of code on different architectures [12]. It allows developers to optimise the resource consumption on (currently) three main levels, namely WCET analysis [12], scheduler optimisation [4] and design pattern based performance optimisation for multi-core processors. First results show a significant reduction in analysis effort while keeping the WCET predictions close above the real WCET with the hybrid method compared to the static and measurement-based approach [12]. However, the source code still needs to be compiled and run on the target hardware with this technique, which still takes quite some time to perform.

3 Early Stage Estimation

Most existing WCET tools perform the analysis on compiled binaries for specific hardware platforms. This approach requires the developers to have a compilable version of the application and the physical hardware platform before any estimate can be given [8] [17] [20]. As a result, it becomes difficult to acquire early insight of the WCET during development. Additionally, hard real-time systems have strict requirements on the WCET as failing deadlines will lead to disastrous consequences. To ensure that all deadlines are met during development, two possible scenarios occur.

On the one hand, system designers assume really pessimistic results of the upper bound to compensate for all errors and assumptions made during analysis. This results in the use of over qualified hardware which will increase the cost of the final product. Whereas small cost savings on better suited hardware will result in huge savings in mass production.

On the other hand, underestimating the final WCET during development leads to financial losses when custom developed hardware does not appear to be sufficient enough to schedule the software tasks [2]. At that point, it is important to "fail fast", so that developers have the opportunity to correct and iterate the design much faster. "Failing faster" will limit development costs as less budget is lost due to unnecessary hardware design time and effort.

The main reason that causes the previous scenarios is the lack of insight on the WCET in the early stages of the development cycle. When we look at the V-model development process, we start with defining the project requirements and add more details to the design with each step, as shown in Figure 1.

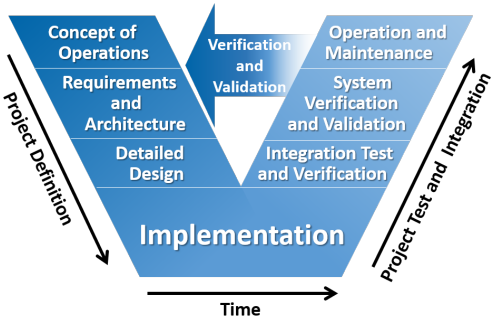

Figure 1 V-model – Development and verification process model.

In the V-model, the code development will only start in the implementation phase at the bottom of the model. As a result, the first opportunity to get insight in the WCET is after the project details are already fixed. In the case of an underestimation of the final WCET, the development process has to move up again in the V-model to adapt the design.

Altenbernd et al. [1] proposed a new methodology to gain early insight into the execution time by "training" a linear time model that translates code into basic instruction of which the timing is determined. Experiments on TACLeBench benchmarks showed promising results [1]. As the model is linear, it becomes rather difficult to model and incorporate non-linear effects on more complex platforms, e.g. caches, pipelines, etc. In addition, each basic instruction needs to be trained by generating and measuring a training program.

In order to obtain faster insight on the WCET, we believe that each system can be characterised right from the start of the development process. The characteristics of a system are described as attributes and represent the system from the high-level design down to the code- and hardware-related components. This would allow software developers to look into the influence of design choices and code changes much faster. However, in the early "*project definition*" stages there is little to no source code available to perform analysis on. Nevertheless, as the project specifications are determined, more and more system attributes are resolved that could hint the software developer to a certain interval in which the WCET is located, e.g. algorithms, code instructions, hardware model, etc. As a result, the system engineers are able to reduce the design space of suitable hardware for the system when making a decision. The key for this problem is to create a predictor to estimate the WCET value according to the system attributes which we try to resolve using machine learning.

4 Machine Learning

The execution time of a software task depends on the instructions of the followed program trace on a specific hardware platform [20]. In the case of the WCET, we are interested in the events and interactions that results in the longest path in time of the software task, such as instructions, input data, pre-emption, caches, etc. All these soft- and hardware characteristics can be described as a collection of attributes for a given software task on a specific platform. As a result, it is possible to develop models with these attributes to make predictions on the WCET. Creating a generic model with classic rules-based programming to assess a given code base for a random platform is nearly impossible. Therefore, we need another approach to create or "*train*" an estimation model with machine learning techniques.

Bonenfant et al. [3] propose a method to approximate the WCET early on in the development by applying machine learning. Their goal is to characterise source code in order to find a formula for a specific target platform and compiler toolchain, which will be achieved

by training a neural network on a set of test programs. The prediction is based on the worst-case event count. A static analysis characterises a program by counting events, which would lead to the worst-case result. The training is performed by matching the worst-case event count with the provided WCET estimates of the test programs. Eventually, the trained network will then predict the WCET of a given program with the event counts from the static analysis and the trained formula of the tested hardware [3]. In addition to the worst-case event counts attributes, the author suggests to make a classification based on the code style attributes, e.g. lines of code, loop nesting, auto-generated or handwritten code, etc.

We believe that machine learning presents a valuable solution to make early WCET predictions by classification of the complex problem statement. However, the approach presented by Bonenfant et al. might suffer from oversimplification [3]. The suggested characterisation by event counting makes a high abstraction from the source code so that valuable information of the code flow get lost. At the end, the code flow and hardware interactions will become too complicated if a program is classified based on the entire code base.

The machine learning approach provides us WCET estimates right from the start of the developed process based on the available system attributes. Therefore, rough estimates with a large deviations are available in the early stages of the development phase. However, the early predictions will provide us insight to explorer the hardware design space and exclude target platforms and configurations which will not be suitable. When the design and development process continues, more accurate attributes becomes available resulting in a gradual improvement of the WCET estimation. At that point, it is important to verify and gain trust in the trained model to obtain sound predictions.

In other research, Griffin D. et al. used a Deep Learning Neural Network (DLNN) to model the influences of other processes on shared resources [7]. The attributes used for the DLNN are the build-in Performance Monitoring Counters (PMC) in the multi-core processor. These counters keep the number of occurred events, e.g. number of cache misses, pipeline flushes, etc. The output is an interference multiplier which will be the worst-case overhead originating from the interferences of other processes. The methodology shows promising results with small underestimation errors on the final WCET [7]. However, this approach still requires a regular WCET analysis on a single core without interference to acquire the final WCET, as the obtained results needs to be multiplied with the interference multiplier. Additionally, the number of PMCs is limited which requires to run multiple measurements on the platform to acquire different parameters [7], which increases the analysis effort.

With the right approach, we are convinced that we can create a methodology based on machine learning which is able to perform accurate WCET predictions for any given architecture by combining/improving the hybrid methodology with machine learning and characterising the source code at lower levels (i.e. hybrid blocks) to avoid oversimplification (e.g. loss of code flow information, etc.) or too complex classifications. In this paper, we will focus on the software related attributes for now.

5 Feature generation

In the first step, we need to derive a set of attributes from the source code. This set of code attributes is used as features to train the machine learning layer and eventually estimate the WCET. As shown in Figure 2, the code related attributes are acquired from blocks that are generated by the *Hybrid Block Generator*. A comprehensive discussion of the block generation with the COBRA framework can be found in [12].

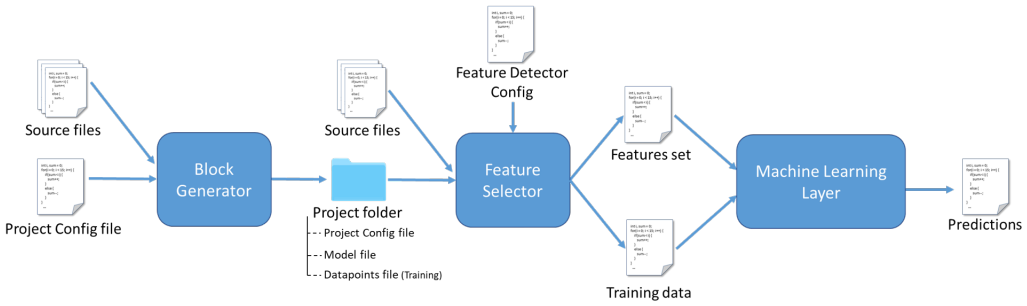

Figure 2 Schematic overview of the Feature Selector in the COBRA-HPA chain.

After generating the hybrid blocks, a "*value*" for each code attribute (i.e. feature) is obtained from these blocks. Extracting these features from source code is a time consuming and error-prone task. Additionally, to train a machine learning layer that is able to provide a "solution", i.e. WCET estimate, we need to apply a supervised learning strategy [6]. As a result, a large annotated training set needs to be created. In order to assist us in analysing and collecting features, we are developing an extension on the HPA-COBRA framework.

The *Feature Selector* module allows us to generate a formatted output file containing all features derived from the hybrid blocks in the project, as shown in Figure 2. The selection of features allows us to describe the characteristics of the code in the blocks at a higher abstraction level. This makes it less complicated to develop and train a machine learning layer that is able to generalise the problem [6]. As stated in Section 4, we need to examine which features have a significant influence on the WCET of code. Therefore, feature design requires adequate insight in the domain to compose a list of potentially relevant, quantifiable features. The next step is to assess the importance of the selected features by checking for correlations between them [9] [10] and eventually training different types of machine learning methodologies to optimise the performance on the verification set.

In order to easily generate and adjust the features for the large hybrid block collection, the *Feature Selector* has a flexible and modular design that enables us to describe a code feature in an XML-file. The detection of features is accomplished by defining and configuring basic detector modules which in turn are chainable to accommodate more complex feature detection rules. In the first prototype of this tool, there are three basic detector modules that already provides an extensive range of possibilities:

- The *Token Count Detector* counts each occurrence of a set of basic tokens and maps the result to the desired feature;
- The *Context Detector* classify if a certain syntactic rule is present in the context of the analysed hybrid block;
- The *Collection Utilities Detector* performs basic set operations on the output of two or more detectors, e.g. accumulate results, union between sets, etc.

The functionality of the basic detectors is built on the open-source parsing framework, ANTLR v4 (*ANother Tool for Language Recognition*). This tool provides the functionality to parse a text file according to a given grammar file [15]. The ANTLR framework is also part of the core of the *Block Generator* tool.

When all attributes of the blocks are determined, a list of features is generated. These features are then ready to be exported to a formatted file. Currently, a CSV exporter module is integrated in the tool which allows us to read the features in a machine learning framework,

Table 1 List of code attributes extracted with the *Feature Selector*.

No. of Additive operations	No. of Multiplicative operations	No. of Division operations
No. of Modulo operations	No. of Logic operations	No. of Bitwise operations
No. of Assign operations	No. of Shift operations	No. of Comparison operations
Return statement present	No. of Evaluation operations	No. of Local variables access
No. of Local array access	No. of Global variables access	No. of Global array access

Table 2 4-Fold cross-validated Mean Relative Error (MRE) for each trained regression model.

Regression models	**MRE average**	**MRE worst-case**
Linear Regression	0.778510	1.385968
Polynomial Regression (2nd Degree)	3.005707	6.693175
Tree Regression	0.338957	0.535293
Random Forest Regression	0.518764	0.846265
Support Vector Regression (Linear Kernel)	**0.272756**	**0.402447**
Support Vector Regression (RBF Kernel)	0.497793	0.839461
K-Nearest Neighbours Regression	0.389732	0.423841
Ridge Regression	0.778263	1.303659

such as Scikit-Learn or TensorFlow. In addition, a WCET-annotated version can be created when the corresponding WCET results are provided. This presents the functionality to generate annotated features for training sets.

6 Experiment

The Hybrid Machine Learning methodology for WCET analysis is a new approach which is currently in the early stages of research. In order to test the functionality and performance, we need to generate data sets to train and validate different machine learning techniques. These data sets are generated from benchmark code of the TACLeBench initiative [5]. The TACLeBench is a benchmark project of the TACLe community to evaluate and compare different timing analysis tools and techniques. The benchmark programs are used by a wide community which has performed timing analysis on different platforms. This provides us a large reference database to train and validate our methodology in later stages.

For this first experiment, we have selected a set of code related attributes which are listed in Table 1. These attributes are modelled in the *Feature Selector* tool, so we are able to easily obtain features from the benchmark code. The attributes in this experiment are selected by visually inspecting the blocks and identifying which code characteristics would have a significant impact on the execution time. For this first prototype, we kept the size of generated hybrid blocks small and the number of features limited. For example, iterations statements were unrolled and not modelled as independent features, as extra features would require more training data which would made the prototype too ambitious.

At the core of the prediction mechanism is a target specific trained machine learning layer. This layer receives a set of features, as the ones in Table 1, and provides one output value that resembles its estimation for the WCET of the hybrid block. The next step is to select a set of machine learning techniques that would be suitable to perform the task. In our case, we need predictors (i.e. features) to predict a numeric value, namely the WCET. This approach is referred to as *regression* [6]. For this experiment, we have trained and evaluated

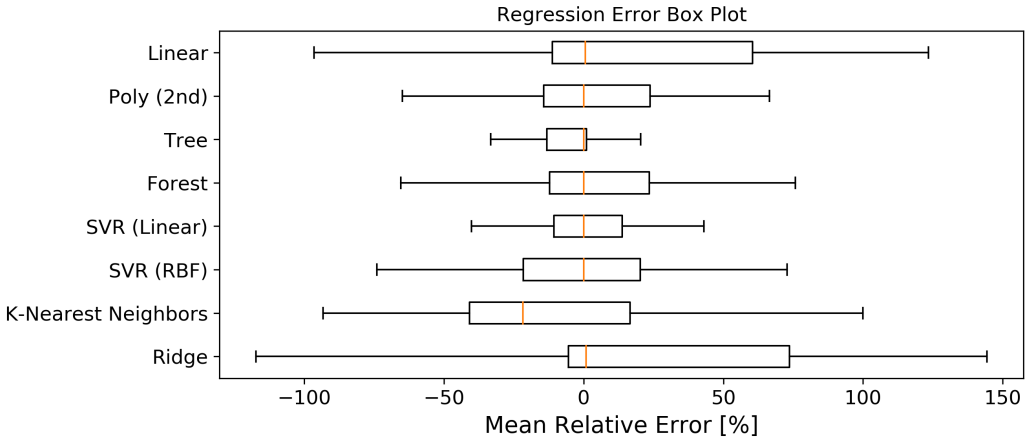

Figure 3 Regression error box plot for each regression model with removed outliers.

a selection of standard regression models, as listed in Table 2. All models in this experiment were implemented with the Scikit-Learn framework [16]. This frameworks provides a wide range of tools to create, train and validate machine learning algorithms in Python.

The training and validation sets are built from hybrid blocks generated by the *Block Generator* tool [12]. The selected blocks originate from benchmarks of the TACLeBench [5]. For this experiment, the training and validation sets have a size of respectively 75 and 25 blocks (datapoints) which are iterated in a 4-fold cross-validation process. Each of those blocks are provided with all attributes from Table 1 and annotated with a WCET value to train the model and verify the results.

The target platform used in this experiment is an ARM Cortex-M3 CPU on the EZR32 Leopard Gecko board of Silicon Labs, where the WCET of each block was measured according to the hybrid methodology explained in Section 2 [12]. To evaluate and compare the performance of the regression models, a formula is needed to get insight in the prediction error. We are mostly interested in the relative error, as an error of 5 clock cycles on big blocks with a total of 10000 clock cycles (+0.05%) is less severe than for smaller blocks with a total of 20 clock cycles (+25%!). Therefore, the Mean Relative Error (MRE) is used in the results for evaluation.

The average and worst MRE scores on the validation sets for each regression model is shown in Table 2. The best performing model in this experiment is the *Support Vector Regression* (SVR) with a linear kernel, followed by *Tree* and *K-Nearest Neighbours* Regression. The graphs in Figure 4 plot the predicted WCET values of the validation set with respect to the corresponding real measured results. The closer a point is vertically located to the dotted line, the smaller the prediction error is. However, the box plots of the error distributions in Figure 3 provides interesting insides. If we remove the outliers, we see that the *2nd Polynomial Regression* actually performs significantly better then initially thought when comparing it to the result of Table 2. As the MRE is sensitive for the large outliers the model predicted, e.g. the model had prediction errors (MRE) down to -4000% and lower!

After validating the ML models on small hybrid blocks, we performed an experiment on three TACLeBench application to test the performance of the hybrid methodology. The results of these experiments are shown in Table 3. This table shows the errors of each model's estimation of the WCET. A negative and positive error resembles respectively an under- and overestimation of the real WCET.

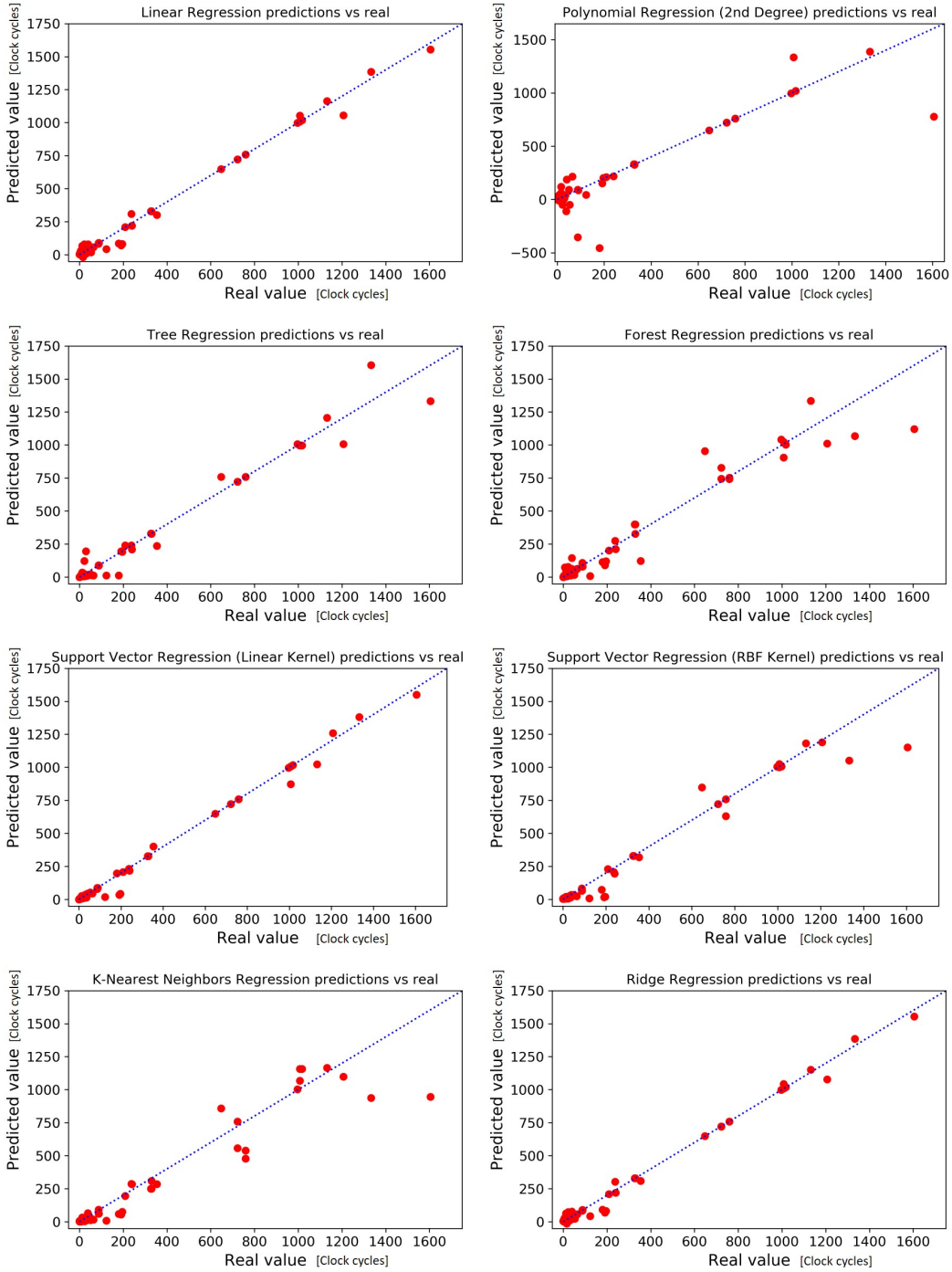

Figure 4 Predicted vs real WCET values for trained regression models on the validation sets.

Table 3 Prediction error of the hybrid ML approach on three TACLeBench application for each trained regression model.

Regression models	Bitonic	Bsort	Recursion
Linear Regression	-49.3%	102.2%	-0.2%
Polynomial Regression (2nd Degree)	100.2%	-266.3%	-10.9%
Tree Regression	18.1%	18%	-52.8%
Random Forest Regression	-11.8%	113.7%	-14.6%
Support Vector Regression (Linear Kernel)	-24%	8.5%	-55.3%
Support Vector Regression (RBF Kernel)	-31.9%	-36.6%	-45.6%
K-Nearest Neighbours Regression	-45,9%	38.5%	-54.1%
Ridge Regression	-47,1%	56.8%	0.5%

The results in Table 3 show good results for the *Tree* and *SVR with Linear Kernel* models. However, they perform significantly worse for the *Recursion* benchmark compared to the other ML models. This benchmark has a small code base which repeatedly gets called recursively. In this case, a small error on a hybrid block will result in a rising total error when the number of recursive calls increases.

7 Discussion and Future Work

In this paper, early stage experiments show that machine learning based WCET prediction is a high potential technique. In the context of the WCET, we are mostly interested in the worst performance as we need to evaluate the error margin in order to obtain the upper bound. The results of the first prototype (Table 2) indicate that the *Support Vector Regression* with a linear kernel has worst-case the lowest MRE of 40.2%, however it reached an average of 27.3%. In addition, we see in Figure 4 that the SVR (Linear Kernel) accurately predicts all features of the verification set with just a few small outliers compared to the other trained regression models. The worst performing cross-validated iteration of the SVR is possibly due to a validation bucket that contained less trained attributes. This problem can be mitigated by further extending the labelled dataset.

The higher performance of the linear kernel models probably is because of the linear characteristics of the trained features on a "simple" architecture, e.g. single core, no caches, etc. Therefore, these models are better in generalising the problem, as more complex models will overfit the solution [6]. The well-performing SVR model tries to fit the data such that the distance between the data points and the fit, which is referred to as the supporting vectors, is maximised [6]. On the other hand, the Tree model partitions the data points in clusters for which a value is assigned to. In this first experiment, we achieved good results. This approach however, estimates discrete values. If we want to have high accurate output values, we need a high partitioning of the data space which results in large complex trees.

Nevertheless, none of the tested machine learning models will be excluded from further research at this point, as these are still preliminary results in this experiment. The results in Table 2 were acquired with the default configuration of each regression model without tuning any of the hyperparameters, as the goal of this experiment is to examine the feasibility of applying machine learning techniques to predict WCET values. Therefore, we can conclude that the WCET estimations show already promising results on small hybrid blocks. We believe that additional tuning of the attributes and models will further improve the accuracy of the predictions.

In future work, we will continue to improve the prediction models [6] [21], selecting the right features in a systematic approach (feature engineering) [9] [10], examine the accuracy/computational complexity trade-off for bigger blocks and extend the list with ensemble models and neural networks. In the case of neural networks, more labelled data is required to train the model in order to achieve acceptable results. In addition, each trained model is platform specific. By including hardware/toolchain related attributes to the model however, we believe a better performing, more generic model can be trained for related architectures. A more general predictor model will lower the effort to train target specific models. In addition, modelling the interferences on shared resources with a DLNN seems a feasible approach as shown by [7]. Therefore, it provides a first step to extend our methodology to more complex hardware.

8 Conclusion

The early stage experiments in this paper show that machine learning-based hybrid WCET prediction is a high potential methodology. In a first stage, we discussed the importance of early stage WCET prediction where current analysis methods falls short, as they rely on an existing code base. In the second stage, we propose to add a predictor model using machine learning to our hybrid methodology. This combined approach is a new concept where code features are utilised to estimate the WCET of a hybrid block.

In order to prove the potential of this methodology, we created the COBRA-HPA framework that splits code into blocks according to the hybrid methodology and dynamically generates corresponding features based on a configurable detector chain. The resulting features can be used to estimated the WCET of the block or to train machine learning models.

Finally, we trained and compared the performance of eight different regression models with code features generated by the *Feature Selector*. The results of this first prototype show that the *Support Vector Regression* with a linear kernel has the best performance. In overall, the WCET estimations of all models have promising results. Additional tuning of the attributes and models will further improve the accuracy of the predictions.

We believe that this approach will be the solution to early stage WCET prediction. Therefore, we will continue to improve the regression models by focussing on feature engineering, tuning prediction models, acquiring more data, extending the models with (deep) neural networks and ensemble models, and integrate hardware/toolchain related features.

References

1 Peter Altenbernd et al. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, Nov 2016. doi:10.1007/s11241-016-9250-7.

2 B. W. Boehm et al. *Software Engineering Economics*. Prentice-Hall PTR, Englewood Cliffs, NJ, 1981.

3 Armelle Bonenfant et al. Early WCET Prediction Using Machine Learning. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57, pages 5:1–5:9, 2017. doi:10.4230/OASIcs.WCET.2017.5.

4 Yorick De Bock et al. Task-Set Generator for Schedulability Analysis using the TACLeBench benchmark suite. In *Proceedings of the Embedded Operating Systems Workshop : EWiLi 2016*, pages 1–6, 2016. URL: http://ceur-ws.org/Vol-1697/.

5 H. Falk et al. TACLeBench: a benchmark collection to support worst-case execution time research. *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET'16)*, 2016.

6 A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017.

7 David Griffin et al. Forecast-based Interference: Modelling Multicore Interference from Observable Factors. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, pages 198–207, 2017. doi:10.1145/3139258.3139275.

8 Jan Gustafsson et al. *Approximate Worst-Case Execution Time Analysis for Early Stage Embedded Systems Development*, pages 308–319. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-10265-3_28.

9 Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

10 Mark A. Hall. Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning. In *Proceedings of the 17th International Conference on Machine Learning*, ICML '00, pages 359–366, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

11 T. Huybrechts et al. Hybrid Approach on Cache Aware Real-Time Scheduling for Multi-Core Systems. In Fatos Xhafa, Leonard Barolli, and Flora Amato, editors, *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 759–768, Cham, 2017. Springer International Publishing.

12 T. Huybrechts et al. COBRA-HPA: a Block Generating Tool to Perform Hybrid Program Analysis. *Int. J. of Grid and Utility Computing*, in press 2018.

13 P. Lokuciejewski and P. Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Netherlands, 2011. doi:10.1007/978-90-481-9929-7.

14 Enrico Mezzetti and Tullio Vardanega. On the Industrial Fitness of WCET Analysis. In *The 11th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2011.

15 Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.

16 F. Pedregosa et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

17 P. Puschner and Ch. Koza. Calculating the Maximum, Execution Time of Real-time Programs. *Real-Time Systems*, 1(2):159–176, 1989. doi:10.1007/BF00571421.

18 Jan Reineke. *Caches in WCET Analysis*. PhD thesis, University of Saarlandes, 2008.

19 Rob van der Meulen. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016, 2017. URL: http://www.gartner.com/newsroom/id/3598917.

20 Reinhard Wilhelm et al. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

21 Dani Yogatama and Gideon Mann. Efficient Transfer Learning Method for Automatic Hyperparameter Tuning. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, volume 33, pages 1077–1085, 2014.