2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC) | 978-1-6654-3420-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/DASC52595.2021.9594326

Deep Neural Network Approach to Estimate Early Worst-Case Execution Time

Vikash Kumar
Computational and Data Sciences
Indian Institute of Science
Bangalore, India
kvikash@iisc.ac.in

***Abstract*—Estimating Worst-Case Execution Time (WCET) is of utmost importance for developing Cyber-Physical and Safety-Critical Systems. The system's scheduler uses the estimated WCET to schedule each task of these systems, and failure may lead to catastrophic events. It is thus imperative to build provably reliable systems. WCET is available to us in the last stage of systems development when the hardware is available and the application code is compiled on it. Different methodologies measure the WCET, but none of them give early insights on WCET, which is crucial for system development. If the system designers overestimate WCET in the early stage, then it would lead to the overqualified system, which will increase the cost of the final product, and if they underestimate WCET in the early-stage, then it would lead to financial loss as the system would not perform as expected.**

This paper estimates early WCET using Deep Neural Networks as an approximate predictor model for hardware architecture and compiler. This model predicts the WCET based on the source code without compiling and running on the hardware architecture. Our WCET prediction model is created using the Pytorch framework. The resulting WCET is too erroneous to be used as an upper bound on the WCET. However, getting these results in the early stages of system development is an essential prerequisite for the system's dimensioning and configuration of the hardware setup.

***Index Terms*—Deep Neural Network, Embedded System, Real-Time System, WCET.**

I. Introduction

In safety-critical systems, the timing domain is as important as the value domain. These systems need to satisfy the timing constraint; otherwise, resource damage or even life loss could occur. For instance, it is essential to know that airbags in cars open fast enough to save lives. Besides, these systems not only satisfy the correctness of the system but also be responsive. If the system does not satisfy the timing constraints after manufacture, then changing the hardware that cannot schedule tasks would be more expensive to redesign. Therefore, estimating the worst-case execution time is very crucial. Estimating WCET [1] for the given architecture is difficult, if not impossible, to cover all the system states, and it requires the user's input. Modern processors are equipped with complex architectural features such as superscalar pipelines and caches that make WCET estimation complex. For instance, caches introduce the variance in operations execution time based on the hit or miss in the caches. In the previous decade, many optimizations have been done to improve the average-case execution time, but less work has been done to estimate WCET precisely and accurately.

The process of estimating the WCET is called timing analysis. The timing analysis of the given system is possible in the last stage of the system development process. Hardware architecture and compiled binary code are required to estimate the WCET. By getting the early estimate of WCET, we will prune the system's unwanted design points based on the parameter of interest, i.e., design system exploration. There are three popular ways of estimating WCET, i.e., the Measurement-based approach, Static Analysis, and the Hybrid approach. The Measurement-based method [2] executes the task on the given hardware or the simulator for the different inputs and states (initial and intermediate) of the architecture to measure the execution time. All different sets of inputs data are applied to measure the maximal program execution time. The measurement-based approach is the most common technique in the industry because hardware and simulators are usually available. The main disadvantage of this method is that determining the inputs to be considered for the WCET is not obvious, and running the WCET analysis over the entire set of possible inputs is not feasible.

On the other hand, the Static approach [3] does not execute the task on hardware or simulator but analyzes the set of possible control flow and reduces the number of different possible inputs using safe abstraction. The success of the static approach is exposed by vendors of the hardware, but in recent years, hardware vendors do not reveal their system features anymore. This approach also has the same shortcomings as the measurement-based approach, in that it overestimates results due to the lack of information. Additionally, the complexity of the static approach leads to increased code size, which is not preferable.

In the Hybrid approach [4], we combine the concepts from the Measurement-based and Static approach. The hybrid approach identifies a single feasible path, a program path consisting of a basic block sequence. The advantage of this approach is that it does not rely on the abstract model of the hardware architecture. However, instrumented code is required, which may not be allowed in all cases, and correct WCET is not possible because safe initial state and worst-

978-1-6654-3420-1/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.

case input can not be assumed. WCET analysis tools such as aiT [5], Chronos [6], Heptane [7], and OTAWA [8] are used to determine the safe upper bound when the hardware architecture and compiled binary code are available. All the existing WCET analysis tools are incapable of predicting early WCET in the system's development process. To overcome this problem, we are proposing an approach to predict early WCET without using the binaries.

In this paper, we have extended the work done in [9]. The linear model presented in [9] will no longer give better accuracy as the system's complexity increases, such as pipeline, caches, and branch prediction. We employ Deep Neural Networks to predict early WCET instead of running the application source code physically on the hardware. We use a Measurement-based approach among various WCET estimation strategies available. To our best knowledge, nobody uses the Deep Neural Network model to estimate early WCET on the available datasets.

The remaining of the paper is structured as follows: Section II presents related work, and in section III, we brief Deep Neural Networks. Section IV describes our WCET estimation approach to predict early WCET using neural networks, which is evaluated in section V. Experimental results are reported in section VI. We conclude the paper finally and present the prospects of future work in section VII.

II. Related work

Bonenfant et al. [10] presented an approach for early WCET prediction using machine learning based on C source code. Their method used a Static approach which generated worst-case event counts such as the number of arithmetic operations like addition, subtraction, multiplication, and division, the number of function calls, the number of global variables, and the number of reads and writes access. To train the model, they used these features with labeled WCET. The worst-case events count of source code was formulated to obtain a satisfiable prediction of the future WCET. As far as considering estimating early WCET, this approach works well. However, it has some shortcomings in that event-counting of code using a Control Flow Graph (CFG) results in the loss of valuable code flow information.

Thomas Huybrechts et al. [11] proposed a new extension to the hybrid approach to predict early WCET using machine learning. This new approach estimates the WCET using 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 into the WCET early on in the development process and refines its estimate when more detailed features are available. A new tool named COBRA was proposed to extract the features. The extracted features were used to train and validate the model. Machine learning approaches, such as Linear regression, Tree regression, and Support vector machines, were used to compare the results of TACLeBench [12] applications. The mean relative error for support vector regression with Linear kernel was 40.2%, which was too high to use as an upper bound on WCET.

Oyamada et al. [13] presented a neural network-based approach for accurate software performance estimation, which also deals with the non-linear behavior of execution times due to complex modern architecture such as deep pipeline, branch prediction, and cache sizes. Assembly instructions were used as features categorized as floating-point, integers, branches, and load/store operations. Based on the CFG, the trained data is classified into two parts as control dominated applications and data dominated applications. Feed-Forward neural networks have been used with one input layer, one hidden layer, and one output layer with different neurons at each layer. CFG weights were used to make the distinction of application domains. The generic estimator had a maximum overestimation of 41.01% and a maximum underestimation of 20.69%. However, for the specialized estimator, they improved the overestimation and underestimation slightly. The error was too high for the estimate to be used as upper bound but obtaining such results in the development process is useful for system design.

The approach proposed in [14] extended the work done in [11] with a deep neural network to estimate WCET. This work used two different models: a feed-forward neural network and a tree recursive neural network. The data used in their experiments was taken from TACLeBench benchmark suits. The architecture used for dataset A was one input layer of ten neurons, three hidden layers, 32 neurons, and one output layer. The results were given in terms of Root Mean Square Error (RMSE), and for dataset A, it was around 40% on validation. The results are too large to obtain any useful upper bound.

Lisper and Santos [15] developed a new Measurement-based WCET analysis method, which uses regression to identify parameters in the common linear Implicit Path Enumeration Technique (IPET) model to calculate WCET. The method can use different granularity timing measurements, including end-to-end measurements, which reduces the need for fine-grained timing measurement instrumentation.

Abel and Reineke [16] developed an algorithm to model the cache's replacement policy by measuring actual hardware automatically. This work helps identify the cache-sensitive timing model.

This paper presents a Deep Neural Network based approach to predict early WCET with a network architecture different from the aforementioned approaches. Our models are evaluated on TACLeBench [12] benchmark suites. We use the SWEET [17] tool to extract the features. The primary function of SWEET is to perform *flow analysis* to identify *flow facts* i.e., information about loop bounds and infeasible paths in the program. Flow facts are necessary for finding safe and tight WCET. Any WCET analysis must satisfy safeness and tightness conditions, which reflects the estimate of WCET precisely.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.

III. Deep Neural Network

Deep Neural Networks [18] are gaining popularity in every field of life due to their ability to solve complicated applications with increasing accuracy over time. They are a subset of Artificial intelligence that attempt to learn patterns based on input data. It is the machine learning techniques that provide computers with the ability to learn from observed data. Supervised learning and Unsupervised learning are different types of machine learning. In supervised learning, the system is given labeled data, whereas in unsupervised learning, the system is given unlabeled data. Our approach uses supervised learning in which labels are formed out of the number of cycles consumed for each training program. This will be further explained in section V.

The analogy of a neural network [18] has been taken from the neurons present in the human brain. The whole concept of deep learning is to try and mimic the human brain and get similar functions as the human brain has and leverage the things that evolution has already developed for us. Millions of neurons are present in the human brain. Neurons send and process signals in the form of electrical and chemical signals. Biological neural networks consist of interconnected neurons with dendrites that receive inputs. Based on these inputs, they produce an output through an axon to another neuron. The neuron (node) is the building block of any deep neural network. An example of a neuron in fig. 1. shows the input (X_1 - X_n), their corresponding weights (W_1 - W_n), a bias (b) and the activation function f applied to the weighted sum of the inputs. The parts/components of a typical deep learning system are described below, and later we apply the following steps to create the model, train the model, and for the accuracy of the model.

- Data - The data is what we apply deep learning techniques on. The data gives insights into how our features and labels are related.
- Task - On the given data, what tasks have to be performed – such as classification and regression.
- Model - Model represents the details of the architecture. Some of the popular models are Feed-Forward Neural Network, Convolution Neural Network, and Recurrent Neural Network.
- Loss Function - The loss function evaluates how well the learning algorithm predicts the outcome. The learning algorithm tries to improve itself by loss functions. There are different types of loss functions such as mean square error and cross-entropy loss.
- Learning Algorithm - The learning algorithm is used to update each parameter in our neural network. Using a learning algorithm, our model learns to identify trends in the data. Some of the different learning algorithms are gradient descent and Adam [19].
- Accuracy - The predicted value is compared with the actual value to find the accuracy which tells us how well our network performs.

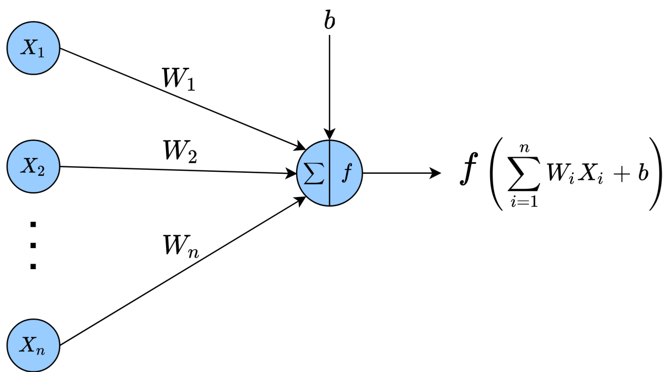


Fig. 1: A Neuron.

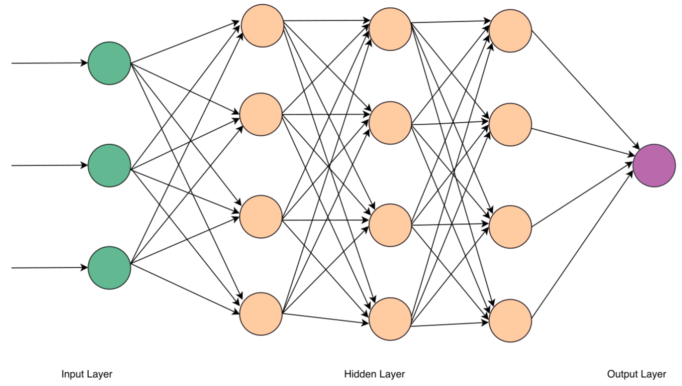


Fig. 2: A Feed Forward Neural Network.

A feed-forward neural network (FFNN) or multilayer perceptron (MLP) [20] is an essential deep learning model. A feed-forward network, as shown in fig. 2. consists of one input layer, one and more hidden layers, and one output layer. In the feed-forward network, neurons are not connected to themselves or neurons in the same layer. A fully connected network is a particular case of a feed-forward network where all neurons of one layer are connected to all the following layer's neurons. In the general case, not all the neurons need to be active, and in some networks, most of them are inactive. Neurons at the hidden layer have two portions – linear, and non-linear activation functions. For the given inputs and weights of each layer, the feed-forward network can predict the desired output. This process is known as a forward pass in deep learning terms. Later, using backpropagation, we update our models' parameters.

IV. Early WCET Estimation using DNN

The detailed explanation of the proposed approach is explained below. Figure 3 depicts both the training and testing phases.

- A selection of training programs is made.
- A training program is compiled and executed on the target hardware or a simulator. The execution time is measured for each program.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.

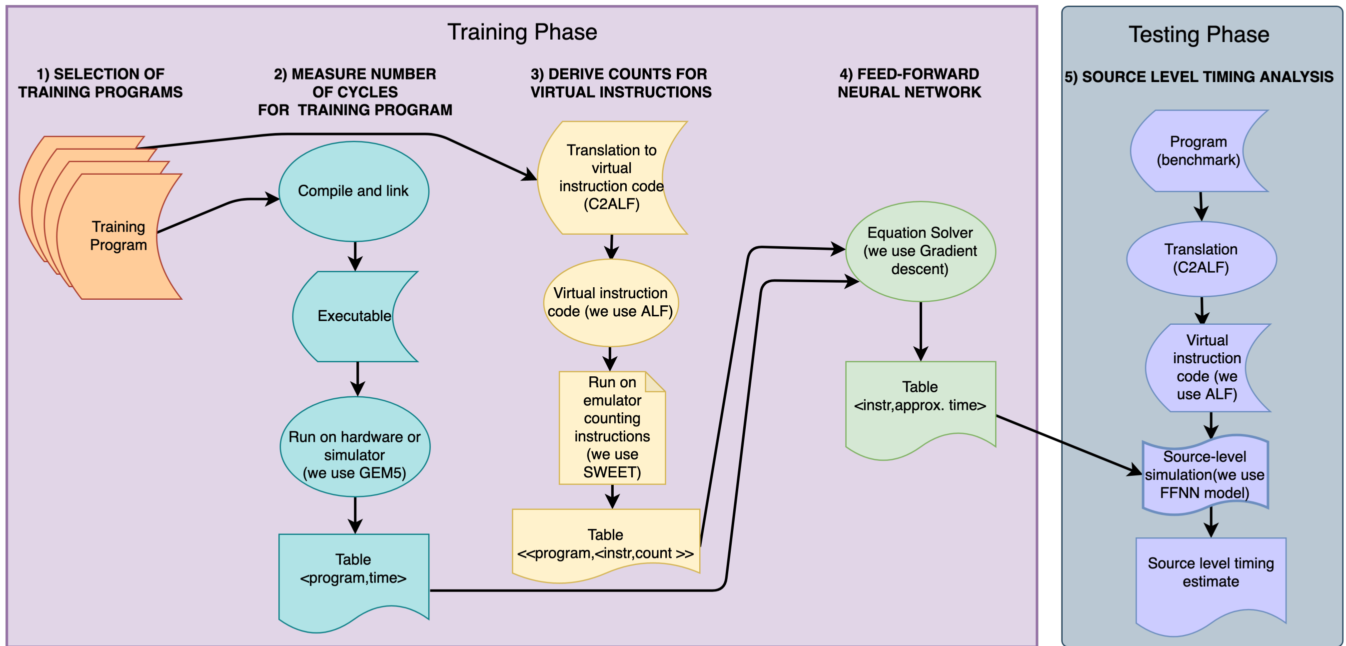


Fig. 3: Early-timing analysis approach.

TABLE I: FEATURES EXTRACTED

Number of addition operations	Number of subtraction operations	Number of multiplication operations
Number of division operations	Number of logic operations	Number of shift operations
Number of function calls	Number of return statements	Number of jump statements
Number of load operations	Number of store operations	Number of comparison operations

- The training program is converted into a virtual instruction set. With the help of SWEET tool, the instruction count of the virtual instructions are recorded.
- The data is fed to the feed-forward neural network. In contrast to work done in [9], this step of WCET estimation is entirely different as they have used a linear model in this step, and this approach uses a Feed-Forward Neural Network.
- Finally, the derived feed-forward neural networks model is used for timing analysis.

V. Experimental Setup

Selection and construction of training programs are of utmost importance. Each training program is constructed using the extended approach presented in [9], which covers all the context-dependent timing effects due to hardware features such as caches, pipelines and branch prediction units, and code optimizations due to the compiler. The training programs are executed on the gem5 [21] simulator to measure the number of cycles which are used as labels. The same training programs are translated into a virtual instruction set using SWEET [17] tool. This virtual Instruction Set Architecture (ISA) acts as a feature set of the proposed predictor network. Combining these two essentially creates the data which can be appropriately used by neural networks.

In deep learning, we need to pre-process and clean the data before feeding it to a neural network. We have used feature selection on our training data, and we found that the features shown in table I are the most frequently occurring out of all the features. So we need to choose the features carefully [22]. We found that some features have values in the range of 200 - 800, and some features have values in the range of 2 - 10. These differences in the range of values bias the model's prediction to be inclined towards values of features that are larger, and the features having lower values contribute much less to the prediction, i.e., low value features have no significance in the Neural Network [23].

To tackle this issue, we need to normalize our data value in the range of 0 to 1. Several frameworks such as Pytorch [24] and Tensorflow [25] are available and provide a competitive arsenal of tools to perform this operation. We have used the Pytorch framework in this experiment. ARTIST2 Language for WCET Flow Analysis (ALF) format is suitable for our approach because it contains both high-level and low-level constructs. Statement such as CALL and RETURN represent high-level constructs, and statements such as LOAD, STORE, and JUMP represent low-level constructs. Two datasets, A and B, are created. The total number of elements in the training datasets A and B are 57 and 224 respectively. 23 TacleBench programs have been used as testing data, and these programs

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.

TABLE II: LAYERS AND PROPERTIES OF OUR NEURAL NETWORK

Dataset A	**Input**	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Properties**	
No. Neurons	12	32	32	32	1	Learning rate / Optimizer	0.01 / Adam
Activation f.	-	Leaky Relu	Leaky Relu	Leaky Relu	Leaky Relu	No. epochs / Batch size	100 / 10
Regularisation	-	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	No. Samples (train / test)	57 / 23
Dataset B	**Input**	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Properties**	
No. Neurons	12	32	32	32	1	Learning rate / Optimizer	0.03 / Adam
Activation f.	-	Leaky Relu	Leaky Relu	Leaky Relu	Leaky Relu	No. epochs / Batch size	100 / 40
Regularisation	-	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	No. Samples (train / test)	224 / 23

are the same as those used in [9]. The training data is further divided into two parts: training, and validation through 5-fold cross-validation to check how well our model performs on the training data. The testing data is taken from the TacleBench benchmark which contains a test case for WCET analysis for different platforms. RMSE scores are used to compare our predicted value to the actual value. RMSE is calculated as the root of the mean of the squared differences between the predictions and the actual values.

Gem5 [21] simulator is used to carry out all experiments to configure one processor with different attributes. The ARM810 processor is used with 5 stage pipeline, 8KB unified cache, MMU, and static branch prediction. Operations like floating points are implemented in the software. LLVM [26] compiler is used for compiling the test programs. Clang is used as a front-end to convert C source code into LLVM intermediate format. The LLVM Intermediate Representation (IR) file is given to ALF backend (C2ALF) to convert it into an ALF format, and the ARM backend (LLC) to convert it into an ARM object file. The SWEET tool is used to read the ALF code; it counts the number of different ALF constructs that appear as statements and operations.

We use hyperparameter tuning [27], such as grid search and randomized search, to find the best possible neural network configuration by modifying the hyperparameters like learning rate (lr), number of epochs, and different optimization and activation functions in other experiments. The best structure of the deep learning model is shown in table II. We executed 12 different experiments by varying all the hyperparameters to find out the model that gives the least error on training and validation data. The comparison of different training and testing loss is shown in fig. 4. The X-axis represents a different experiment with hyperparameter tuning and the Y-axis loss values. In Figure 4(a)., the experiment with lr = 0.01 and neurons = 32 is the best one for dataset A as it gives the least error on both training and validation data. The training loss error is significantly less than the validation loss error because dataset A has a considerably smaller sample program. Similarly, the experiment with lr = 0.03 and neurons = 32 is the best for dataset B. Learning curve of the best model for both datasets is shown in Figure 5. The loss scale is different in both figures because of the different numbers of the samples in each dataset. The figure indicates some oscillation in loss values at the beginning for dataset A and is smooth for dataset B, but as the number of epochs increases with time, loss values start converging and saturate to zero. Hence, we limit the number of training epochs to 100.

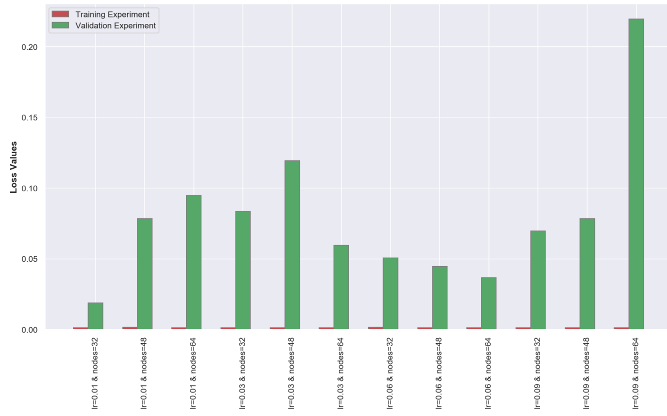
VI. Results

The results are shown in the graph in Figure 6; the percentage RMSE in table III provide better insights into the model. We have executed our Deep learning model 12 times with each combination of different hyperparameters values. This allows us to calculate the minimum, maximum, and average error values for each configuration set up. The minimum, maximum, and average values are shown in Figure 6. For dataset A, we notice that the results with *lr = 0.01* and *nodes = 32* have shown better results where the minimum and average RMSE values are very close. Also, the maximum RMSE is comparatively close to these values. The trend in dataset B is different as the results with *lr = 0.03* and *nodes = 32* have shown the lowest error rate. Although there is variation in results, there is not a big difference between the minimum and maximum error values, which shows that, in most cases, our model has lower error rates as compared to few cases where the error rate is high. The large error is due to the considerable difference between the properties of training and testing datasets.

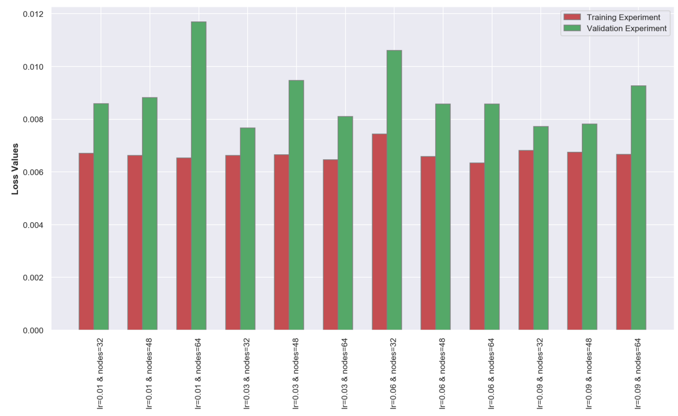
We compare our method with other methods in the literature by also pointing out similarities and differences between the two:

- We use statements and operations of source programs as a feature, similar to the approaches presented in [10, 11, 13, 14].
- Unlike [10] and [11], we do not use a static approach. Measurement-based approach is used in a target-hardware agnostic manner.
- Unlike [13] we do not use assembly instructions for feature categorization. We have used a source program to extract the features.
- Like in [14] we have used Root Mean Square Error (RMSE). However, our results are different because we have used different datasets and WCET strategies.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.

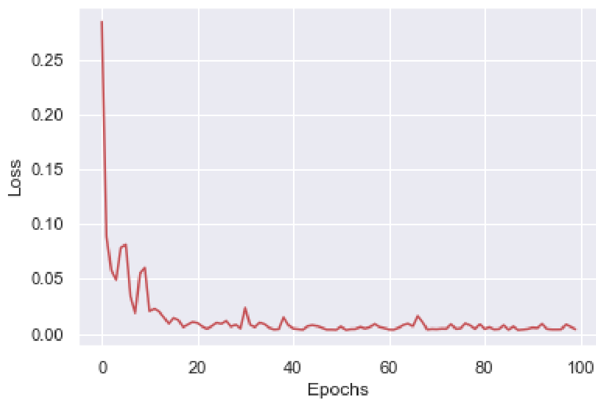


(a) Dataset A

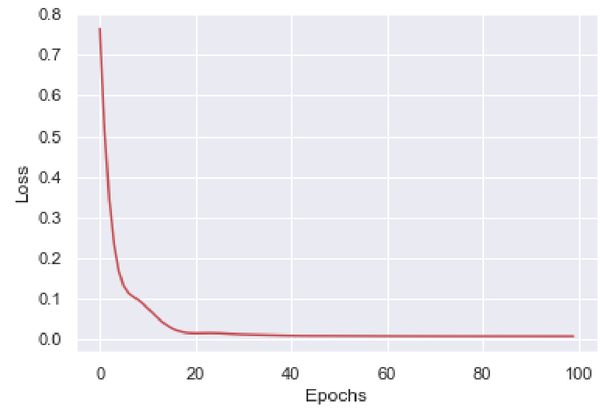


(b) Dataset B

Fig. 4: Comparison of Loss values using different configuration.

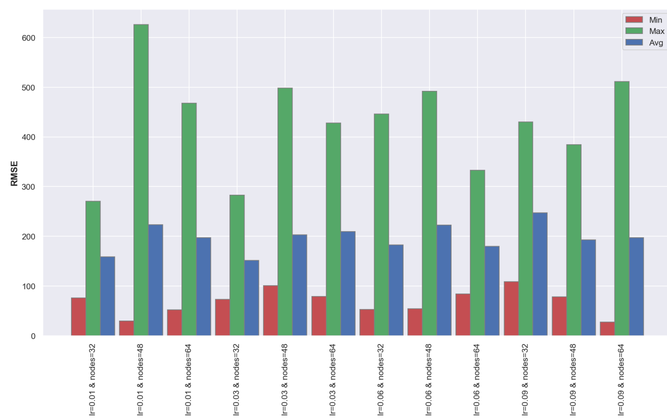


(a) Dataset A

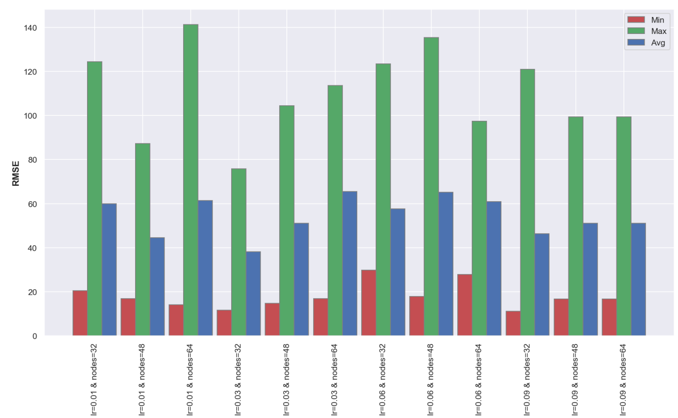


(b) Dataset B

Fig. 5: Comparison of Learning Curve.



(a) Dataset A



(b) Dataset B

Fig. 6: Comparison of Min, Max, and Avg RMSE values by executing the model using 12 different configuration settings.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.

TABLE III: RMSE ERRORS OF NEURAL NETWORK MODELS.

Dataset	Average Error (RMSE) Training / Test	Min. Error (RMSE) Training / Test	Max. Error (RMSE) Training / Test
A	21% / 41.3%	17.8% / 23%	22.9% / 66.7%
B	12.7% / 20.6%	11.8% / 17.4%	14.8% / 23.5%

VII. Conclusions and Future Work

This paper presents an approach that can be used to predict early WCET using a Deep Neural Network. The model estimates WCET from the source code of the applications. Features are generated using the SWEET tool, i.e., the number of statements and operations in the source code, which are fed into our networks to predict WCET after scaling data. Two datasets, A and B, are created with 57 and 224 samples respectively. We demonstrate the model for ARM processors. We have used the Pytorch framework to implement a feed-forward neural network that converges quickly. The model performance is evaluated using a metric called the RMSE. We calculated the minimum, maximum, and average RMSE for each distinct neural network configuration. The RMSE value for the bigger dataset is much better than the smaller dataset. The results shown in this paper are not promising as it is too large to use as an upper bound. However, getting these numbers in the early stage of developing a system is useful in many ways, such as preventing systems designers from assuming a pessimistic upper bound on the WCET.

We believe that Deep Neural Networks can be applied to improve these results further. Instead of using a simulator for measurement, we can use a real processor to obtain more accurate models. This is a subject for future research, too. We intend to train deeper models on bigger datasets to obtain even better and reliable results in the future. Deep learning has good potential in WCET analysis. Additionally, we can further reduce WCET analysis results' pessimism by applying different models like Convolution Neural Network (CNN) and Recurrent Neural Network (RNN) that can be used to estimate early WCET.

Acknowledgment

The authors would like to thank Sourav Mishra for his valuable suggestions on several aspects of this paper. We would also like to thank the anonymous reviewers for their insightful comments and feedback.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[2] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based worst-case execution time analysis," in *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*. IEEE, 2005, pp. 7–10.

[3] R. Heckmann and C. Ferdinand, "Worst-case execution time prediction by static program analysis," in *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004, pages 26–30. IEEE Computer Society*, 2004.

[4] S. M. Petters, "Bounding the execution time of real-time tasks on modern processors," in *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*. IEEE, 2000, pp. 498–502.

[5] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," in *Building the Information Society*. Springer, 2004, pp. 377–383.

[6] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 56–67, 2007.

[7] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[8] H. Cassé and P. Sainrat, "Otawa, a framework for experimenting wcet computations," 2006.

[9] P. Altenbernd, J. Gustafsson, B. Lisper, and F. Stappert, "Early execution time-estimation through automatically generated timing models," *Real-Time Systems*, vol. 52, no. 6, pp. 731–760, 2016.

[10] A. Bonenfant, D. Claraz, M. De Michiel, and P. Sotin, "Early wcet prediction using machine learning," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[11] T. Huybrechts, S. Mercelis, and P. Hellinckx, "A new hybrid approach on wcet analysis for real-time systems using machine learning," in *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[12] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen,

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.

P. Wägemann, and S. Wegener, “Taclebench: A benchmark collection to support worst-case execution time research,” in *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[13] M. S. Oyamada, F. Zschornack, and F. Wanger, “Accurate software performance estimation using domain classification and neural networks,” in *Proceedings. SBCCI 2004. 17th Symposium on Integrated Circuits and Systems Design (IEEE Cat. No. 04TH8784)*. IEEE, 2004, pp. 175–180.

[14] T. Huybrechts, T. Cassimon, S. Mercelis, and P. Hellinckx, “Introduction of deep neural network in hybrid wcet analysis,” in *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. Springer, 2018, pp. 415–425.

[15] B. Lisper and M. Santos, “Model identification for wcet analysis,” in *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2009, pp. 55–64.

[16] A. Abel and J. Reineke, “Measurement-based modeling of the cache replacement policy,” in *in 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS*, 2013, pp. 65–74.

[17] B. Lisper, “Sweet–a tool for wcet flow analysis,” in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2014, pp. 482–485.

[18] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[19] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[20] D. Svozil, V. Kvasnicka, and J. Pospichal, “Introduction to multi-layer feed-forward neural networks,” *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[22] I. Guyon and A. Elisseeff, “An introduction to variable and feature selection,” *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[23] M. A. Hall, “Correlation-based feature selection of discrete and numeric class machine learning,” 2000.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[26] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[27] M. Feurer and F. Hutter, “Hyperparameter optimization,” in *Automated Machine Learning*. Springer, Cham, 2019, pp. 3–33.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 18,2023 at 11:04:55 UTC from IEEE Xplore. Restrictions apply.