Efficient ML-Based Performance Estimation Approach across Different Microarchitectures for RISC-V Processors

Weiyan Zhang[1] Mehran Goli[2] Muhammad Hassan[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
weiyan.zhang@dfki.de {mehran, hassan, drechsler}@uni-bremen.de

***Abstract*—High-level performance estimation using *Machine Learning* (ML) can significantly facilitate the exploration of a wide range of processor microarchitecture solutions at the early stage. Moreover, for the selected microarchitecture, it can remarkably accelerate the software optimization step. Recently, ML has been successfully applied to estimate performance, in particular the clock cycles, for various microarchitecture implementations. However, this is clearly not sufficient as the modern processor microarchitectures are complex and require deeper insights into microarchitectural behaviors for better high performance estimation. In this context, finding an accurate and fast approach that can support performance estimation of various microarchitecture implementations of RISC-V *Instruction Set Architecture* (ISA) is very challenging.**

In this paper, we go beyond performance estimation based on clock cycles, i.e., we expand on ML techniques to estimate microarchitectural behaviors. We propose a novel approach based on ML to estimate the performance of embedded software on RISC-V processors across different microarchitectures. Our approach leverages a fast functional simulator, cycle-accurate *Register Transfer Level* (RTL) implementations, and ML techniques to generate *Predictive Models* (PMs) that provide accurate performance estimation while maintaining fast simulation time. In addition to measuring the clock cycles, we also provide insights into the microarchitectural behavior of different microarchitectures by estimating cache misses/hits, branch prediction behavior, and memory dependencies. Experimental results on four real-world cycle-accurate implementations of RISC-V ISA with different microarchitectures at RTL show that using the proposed approach leads to a huge performance boost up to 2261.4× compared to RTL simulations with an average prediction error 0.4%.

***Index Terms*—Performance estimation, Machine learning, Embedded software**

I. Introduction

In recent years, embedded systems have become more central to automation and *Internet of Things* (IoT). They are becoming more complex as the required functionality increases and silicon technology becomes more advanced. As a result, there is a growing need for open-source solutions that can keep pace with this rapid development. RISC-V, a free and open-source *Instruction Set Architecture* (ISA), has shown enormous potential, in particular, for embedded systems used in various utilization domains. RISC-V is characterized by its modular and extensible design, empowering designers to create customizable processor implementations that can be tailored to a wide variety of applications. Various RISC-V implementations at different levels of abstraction are available in its ecosystem, making it a versatile choice for a wide range of applications.

Performance, measured in terms of the number of clock cycles, is a critical design constraint that needs to be profiled in system design. Traditionally, designers often rely on different tools to simulate and analyze the performance impact resulting from various design trade-offs, including system components, interconnect, and memory layout. There are various performance estimation tools implemented at different levels of abstraction, ranging from high-speed functional simulator that lack cycle information to low-speed *Register Transfer Level* (RTL) implementations that provide accurate cycle information. Although RTL enables accurate performance measurement and design optimization, slow RTL simulation often becomes the bottleneck of the design process. To effectively explore potential options for different design choices, a robust performance estimation approach is needed.

Machine Learning (ML) is a popular technique that is extensively utilized in various fields, including embedded system design. It has demonstrated the capacity to effectively address both linear and nonlinear problems. Traditionally, architectural and system optimizations were typically performed to speed up the execution and enhance the performance of ML models. However, there are now indications of utilizing ML in a different way: to improve the design of embedded systems themselves [1], [2]. This trend is challenging traditional methods of system design and creating new opportunities for optimization and innovation. For embedded systems, the execution process of hardware can be modeled by trained *Predictive Models* (PMs). The basic idea is to collect the reference executed cycles and performance-related parameters, such as dynamic instruction counts, during hardware execution, and apply ML algorithms to train the models. The performance-related parameters are selected in such a way that the approach has the lowest dependency on microarchitectural and software details. To predict the performance of new software, a fast functional simulator is used to quickly obtain their performance-related parameters, which are then applied to the trained PMs.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within projects Scale4Edge under grant no. 16ME0127, ECXL under grant no. 01IW22002 and VE-HEP under grant no. 16KIS1342.

Recently, [2] used ML techniques to predict the clock cycles. However, this is clearly not sufficient as the modern processor microarchitectures are complex and require deeper insights into microarchitectural behaviors for better high performance estimation.

In this paper, we propose an ML-based methodology to estimate the performance of embedded software for RISC-V processors. Considering that the real physical hardware may not be available, we employ a combination of fast functional simulator, cycle-accurate RTL implementations, and ML techniques to generate PMs. Subsequently, the PMs are combined with a fast functional simulator to enable fast and accurate prediction of the performance of new embedded software. We demonstrate the generalizability of our approach across multiple microarchitectures for the RISC-V ISA by applying it to four real-world, cycle-accurate RTL implementations of the RISC-V ISA. We evaluate the performance of our approach on a set of standard benchmarks from TACLeBench [3]. Our results show that our approach is highly effective, resulting in a remarkable speedup of up to three orders of magnitude, surpassing traditional RTL simulations. Additionally, our predictions exhibit high accuracy in estimating clock cycles. Furthermore, our approach has been shown to be scalable, allowing for estimation of microarchitectural behaviors such as branch prediction, memory dependency prediction, cache miss/hit, LSU busy (i.e., the number of cycles waiting for data memory), and fetch wait (i.e., the number of cycles waiting for instruction fetches), which play a crucial role in achieving high performance in many modern microarchitectures.

The paper is structured as follows. Section II gives an overview of previous works on performance estimation. The proposed methodology is introduced in section III. Experimental results to show the effectiveness of our approach are presented in section IV. Finally, section V concludes this paper and provides an outlook.

II. Related Work

A. Performance Estimation

Performance estimation techniques mimic the behavior of real hardware, they always make a trade-off between simulation accuracy and speed. They can be divided into two main categories: simulation and analytic-based models.

Simulation based approaches model system architectures at different levels of abstraction. Functional simulator such as [4] allows fast prototyping but lack of precision. At the *Electronic System Level* (ESL) [5], SystemC-based *Virtual Prototype* (VP) is often used before the detailed hardware implementation is finalized. This abstraction gives some speedup over cycle-accurate modelling at a low abstraction level. RTL simulation offers a high degree of accuracy in verifying the functionality of digital circuit designs. It can detect errors at an early stage, leading to reduced cost in the design process. However, its simulation speed is comparatively slow. In addition, performance profilers are employed to evaluate the performance of embedded systems at different levels of abstraction by analyzing the collected runtime data during the execution. For instance, Prof5 [6] estimates time and power consumption by modeling the per-cycle power and execution cycles of all instructions. But it does not have the ability to predict more complex architectures like out-of-orders and branch predictors.

To effectively explore potential options for different choices of processor, performance estimation of embedded software at a higher level of abstraction is necessary. Analytic-based models are another option and may be linear [7]–[10] or nonlinear [2], [11], depending on the modeling approach used to express the model. It is easy to implement and efficient to train. During the system design process, analytical models are highly suitable for rapid performance prediction of embedded systems. The main advantage of this approach is that system designer can apply analytical models to quickly perform estimation without requiring knowledge of the specific behavior of software and architectural details. *Linear Regression* (LR) is one of the most widely used algorithms for estimating the performance. In [12], the authors have used analytical models to estimate the execution time of software component on a specific architecture. The paper [2] proposes a learning-based modeling approach that utilizes *Artificial Neural Network* (ANN) models based on information extracted from VP to estimate the number of clock cycles at a higher level of abstraction. An important advantage of the ANN model is that it can easily capture the nonlinear behavior of the processor. The learning-based approaches are comprised of two phases, training and prediction. The model is trained using Matlab or Python on the host machine. Analytic-based approaches have been verified using PowerPC 750 [11], Intel i960KB processor [13], ARM processor [8], LEON3 processor [7] and RISC-V processor [2] as target architectures. However, existing models and approaches for performance analysis may not fully capture the complexity of microarchitectural behaviors, such as cache behavior and branch prediction. Furthermore, the generalizability of the proposed techniques to target processors beyond the testing conducted in each respective paper has not been fully established by the authors.

Our work utilizes analytical modeling as the foundation. While previous research in this field predominantly employed LR or ANN, we aim to compare their performance with newer techniques. Specifically, we investigate the extent to which other regression algorithms can provide more accurate predictions. To achieve this, we implement multiple supervised learning algorithms for performance prediciton, including four classic ML algorithms: *Ordinary Least Squares* (OLS) regression, LR with *Mini Batch Gradient Descent* (MGD), ridge regression and ANN. OLS regression and LR with MGD are LR techniques that optimize the model's parameters differently. Ridge regression is a regularized form of LR that prevents overfitting. ANN is a more complex model capable of capturing nonlinear relationships among variables.

B. RTL-based RISC-V Implementations

The advantage of RTL is that it is essential for the hardware design development process and by definition, it is 100% accurate. There are various RTL implementations available for RISC-V, including many open-source cores that are easily accessible. In the following, we exemplarily review some of

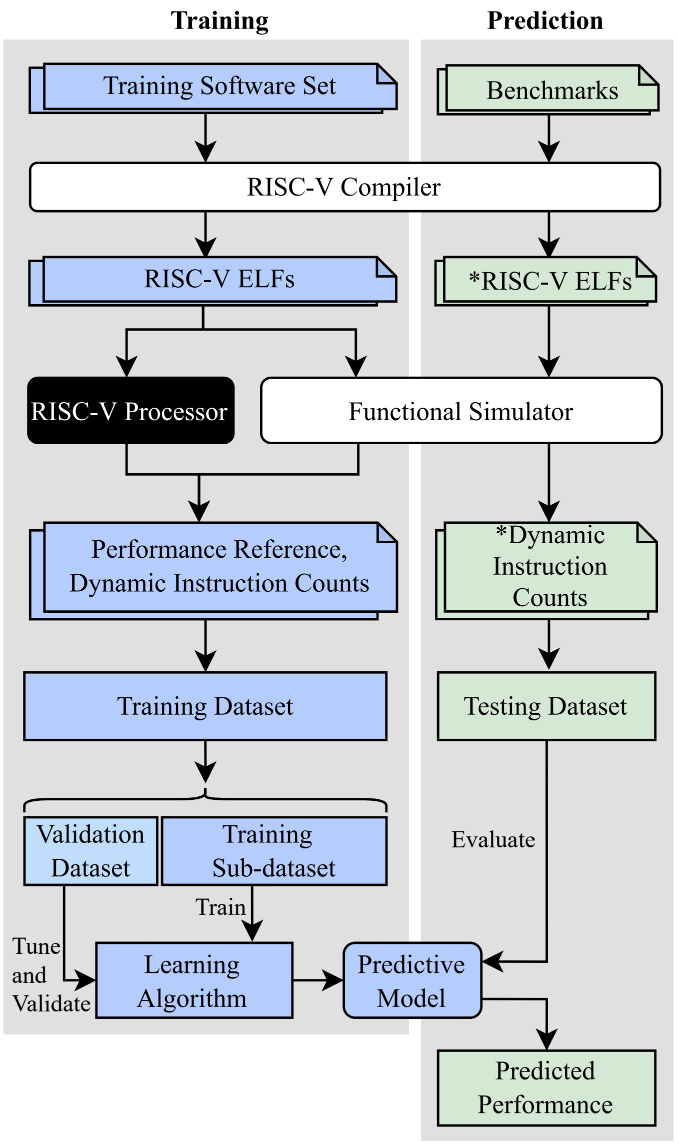

Fig. 1. Performance estimation workflow.
* indicates that RISC-V ELFs and dynamic instruction counts are for benchmarks.

these cores. For example, Ibex [14] is programmed using SystemVerilog, which is highly parametrizable and well-suited for embedded control applications. RSD [15] is a 32-bit RISC-V out-of-order superscalar processor core written in SystemVerilog. SweRV [16] is also programmed using SystemVerilog and allows up to two instructions per clock cycle. Additionally, Western Digital has open-sourced the SweRV *Instruction Set Simulator* (ISS) called Whisper [4] along with the SweRV Core. μRV32RTL [17] is implemented in the modern Scala-based SpinalHDL and suitable for FPGA synthesis. To verify the generalizability of our approach, our PMs are generated based on these four cores.

III. Proposed Methodology

In this section, we present our proposed approach for performance estimation, which involves coupling a fast functional simulator with ML-based models. An overview of our approach is depicted in Fig. 1. The foundation of our methodology lies in supervised learning, a widely utilized ML technique. It encompasses two crucial phases: the training phase and the prediction phase. In the subsequent subsections, we will delve into a detailed explanation of each phase, outlining their significance and key components.

TABLE I
Performance Counters Used in Prediction.

Core	Language	Predicted Counters
Ibex	SystemVerilog	Cycles, LSU Busy, Fetch Wait
RSD	SystemVerilog	Cycles, IC misses, DC load misses, DC store misses, branch prediction misses, memory dependency prediction misses
SweRV	SystemVerilog	Cycles
μRV32RTL	SpinalHDL	Cycles

Abbreviations: IC - Instruction Cache, DC - Data Cache.

A. Predictive Model Training

During the training phase, a set of programs (which we denoted as the "training software set") is prepared. To generate the RISC-V binaries on our host computer, we take advantage of the cross-compilation. A set of training software is compiled with the RISC-V GNU Compiler Toolchain [18] to generate the *Executable and Linkable Formats* (ELFs). The ELFs of training software are then executed on a fast functional simulator to obtain the dynamic information using built-in instruction counters and on a RISC-V processor to obtain the reference executed cycles and microarchitectural behaviors from performance counters. The RISC-V processor could be either a real physical hardware, such as a development board, or a cycle-accurate implementation at RTL, if physical hardware is not available. In this paper, four RISC-V implementations at the RTL were considered to show that our approach is generalizable across the RISC-V microarchitectures. These cores are simulated using Verilator [19], which is an open-source Verilog/SystemVerilog simulator. Table I shows the programming language and obtained performance counters for each core. These performance counters will be predicted in the next phase and can be seen as outputs of our PMs. Among these four cores, only μRV32RTL is written in SpinalHDL [20]. In addition to simulate Verilog and SystemVerilog, Verilator is also supported as a backend for SpinalHDL.

The training dataset consists of the dynamic instruction counts, the reference executed cycles and the other microarchitectural behavior counts (if available), which are extracted from runtime information. Dynamic instruction counts can be seen as performance features for accurately and succinctly capturing and representing program execution. For each program in the training dataset, it is represented as a feature vector. Let N be the total number of programs in the training software set. The input matrix $\mathbf{X} = [\mathbf{x}_1^{\mathrm{T}}, \cdots, \mathbf{x}_N^{\mathrm{T}}]^{\mathrm{T}}$ and the output matrix $\mathbf{Y} = [\mathbf{y}_1^{\mathrm{T}}, \cdots, \mathbf{y}_N^{\mathrm{T}}]^{\mathrm{T}}$ are defined, where $\mathbf{x}_i \in \mathbb{N}^{1\times U}$ denotes the feature vector containing U features for program i and

$\mathbf{y}_i \in \mathbb{N}^{1\times V}$ represents its corresponding vectorized V outputs obtained from the performance counters of the target processor. The training dataset is then represented as

$$D_{train} = \{d_i | d_i = \{\mathbf{x}_i, \mathbf{y}_i\}; 1 \leq i \leq N\}. \tag{1}$$

A cross-validation technique on the training dataset is adopted to help us find the best parameters for our PMs and prevent overfitting. The training dataset is divided into a training sub-dataset and validation dataset. Furthermore, OLS regression, LR with MGD, ridge regression and ANN are applied to the generated training sub-dataset, and the validation dataset is used to tune and validate the training models to build PMs. The performance of each algorithm is compared to find the best suited algorithm.

B. Performance Prediction

In this phase, the PM is tested by a set of new software selected from standard benchmarks. The benchmarks are compiled to generate the ELF files. Subsequently, each ELF file is executed on a functional simulator to obtain dynamic instruction counts (i.e. performance features). For a given testing software set with M benchmarks, let $\hat{\mathbf{X}} = [\hat{\mathbf{x}}_1^{\mathrm{T}}, \cdots, \hat{\mathbf{x}}_M^{\mathrm{T}}]^{\mathrm{T}}$ denote the set of performance feature vectors, where $\hat{\mathbf{x}}_j \in \mathbb{N}^{1\times U}$ represents the performance feature vector for benchmark j. The testing dataset consists exclusively of performance features, namely:

$$D_{test} = \{d_j | d_j = \{\hat{\mathbf{x}}_j\}; 1 \leq j \leq M\}. \tag{2}$$

The testing dataset is used as inputs to the PM to estimate the clock cycles and microarchitectural behaviors. The overall prediction for the whole testing set is represented as $\hat{\mathbf{Y}} = [\hat{\mathbf{y}}_1^{\mathrm{T}}, \cdots, \hat{\mathbf{y}}_M^{\mathrm{T}}]^{\mathrm{T}}$, where $\hat{\mathbf{y}}_j \in \mathbb{N}^{1\times V}$ denotes the corresponding vectorized performance counters for benchmark j.

To evaluate the performance of our PM, we calculated the *Absolute Percentage Error* (APE) metric for each performance counter of every benchmark, defined as

$$APE = \left| \frac{y - \hat{y}}{y} \right| * 100\%, \tag{3}$$

where y and $\hat{y}$ are considered as the real and estimated values of the performance counter for each benchmark.

By calculating the APE values, we were able to quantify the extent of deviation between the real and estimated values for each performance counter. To provide a more comprehensive evaluation, the *Mean Absolute Percentage Error* (MAPE) is computed. This involves summing up the APE values corresponding to each performance counter and dividing the sum by the total number of benchmarks, i.e.,

$$MAPE = \frac{\sum_{j=1}^{M} APE_j}{M}, \tag{4}$$

where APE_j is the APE of benchmark j. The resulting MAPE value represents the average percentage deviation between the predicted and actual values across all benchmarks.

C. Machine Learning Algorithms

We employed supervised ML algorithms to predict the performance of new software. The accuracy of our approach is dependent on the choice of the ML algorithm employed. To find the most suitable algorithm for the target processor, various ML algorithms are outlined below.

1) Ordinary Least Squares Regression: LR models the linear relationship between features and results. OLS is a LR technique that is used to estimate the coefficients of LR models. Define coefficient matrix $\mathbf{W} = [\mathbf{w}_0^{\mathrm{T}}, \cdots, \mathbf{w}_V^{\mathrm{T}}]^{\mathrm{T}}$ and the augmented matrix $\mathbf{X}_{aug} = [J_{N_{sub}\times 1} \mid \mathbf{X}_{sub}]$, where $\mathbf{w}_0$ denotes the intercept vector, N_{sub} is the number of training software used to generate the training sub-dataset, $J_{N_{sub}\times 1}$ is an all-ones matrix with dimension $N_{sub} \times 1$, and $\mathbf{X}_{sub}$ consists of feature vectors in the training sub-dataset. We want to find the best coefficient matrix $\hat{\mathbf{W}}$ that allows the training sub-dataset to fit

$$\mathbf{Y}_{pre} = \mathbf{X}_{aug}\mathbf{W}, \tag{5}$$

where $\mathbf{Y}_{pre}$ is the predicted output matrix.

The problem in (5) can be solved by minimizing the residual sum of squares between the real values in the dataset and predicted values by the linear approximation. The solution is given by

$$\begin{aligned} \hat{\mathbf{W}} &= \underset{\mathbf{W}}{arg\,min} \left\| \mathbf{X}_{aug}\mathbf{W} - \mathbf{Y}_{sub} \right\|_F^2 \\ &= (\mathbf{X}_{aug}^{\mathrm{T}}\mathbf{X}_{aug})^{-1}\mathbf{X}_{aug}^{\mathrm{T}}\mathbf{Y}_{sub}, \end{aligned} \tag{6}$$

where $\mathbf{Y}_{sub}$ is the output matrix in the training sub-dataset, and $\|\cdot\|_F$ is the Frobenius norm. After solving the coefficients, the LR model that can be used for prediction has the following form:

$$\mathbf{y} = [1 \mid \mathbf{x}]\hat{\mathbf{W}}, \tag{7}$$

where $\mathbf{y}$ is the predicted performance counter vector for a given software and $\mathbf{x}$ is the corresponding feature vector.

2) Linear Regression with Mini Batch Gradient Descent: Gradient Descent (GD) is the process of minimizing computational complexity and optimizing a loss function to find the coefficients in an iterative way that corresponds to the best fit between predicted values and actual values. MGD is a variant of the GD algorithm and is used to calculate model error and update model coefficients by splitting the training sub-dataset into small batches. The loss function – the *Mean Squared Error* (MSE) – is minimized during the training period.

3) Ridge Regression: Ridge regression is an extension of LR where the loss function is modified by adding a penalty parameter, known as ridge coefficient. Therefore, the problem to find the best coefficients becomes:

$$\underset{\mathbf{W}}{min} \left\| \mathbf{X}_{aug}\mathbf{W} - \mathbf{Y}_{sub} \right\|_F^2 + \alpha \left\| \mathbf{W} \right\|_F^2, \tag{8}$$

where α is the tuning parameter that controls the shrinkage of the penalty.

```
// Generate a specified number of SLTI instructions
// based on the value of num_slti
void slti_gen(int num_slti) {
  int i = 0;
  while (i < num_slti) {
    bool slti = (i < 3);
    i++;
  }
}
```

Fig. 2. Training software module generating the SLTI instruction.

4) Artificial Neural Network: To capture the nonlinear behavior of software performance, feedforward and backpropagation ANN is used to create performance PM. ANN is composed of an input layer, any number of hidden layers, and an output layer. Each layer may have a different number of neurons and different activation functions. The matrix $\mathbf{X}_{sub}$ is fed into the input layer and sent forward through different connections from one neuron to another in the network. Each neuron holds a number bias and has an activation function, and each connection holds a weight. Once they reach the output layer, the estimated performance counters are obtained. Backpropagation is for calculating the gradients efficiently. It always starts at the output layer and propagates backward, updating weights and biases for each layer in order to produce the desired output at the output layer.

The architecture of an ANN is determined by a set of hyperparameters. These hyperparameters play a crucial role in shaping the network's structure and behavior. In order to identify the optimal hyperparameters, a technique known as hyperparameter tuning is employed. This technique allows for the exploration of a range of hyperparameter configurations and facilitates the selection of the most suitable model architecture. During our experiments, we focused on tuning five key hyperparameters: the learning rate, the number of hidden layers, the number of neurons in each layer, the activation function in each layer, and the number of epochs. Hyperparameter tuning involves systematically exploring various combinations of these hyperparameters to find the configuration that results in the best-performing model. The goal is to minimize the error and improve the accuracy of the model. To achieve this goal, *Adaptive Moment Estimation* (Adam) optimizer is used to minimize the MSE. The PM is generated when Adam converges to the optimal solution.

IV. Experimental Evaluation

A. Experimental Setup

In the training phase, we took advantage of about 700 programs as training software set which were generated by providing different inputs to self-written sample programs and several standard benchmarks from TACLeBench [3]. Fig. 2 shows the module for generating the SLTI instruction in the self-written sample program. The number of SLTI instructions can be controlled by a specified value. During compilation, RISC-V compiler was configured as RV32I. RV32I contains 40 instructions, except for EBREAK, which is used to return control to the debugging environment, we considered the remaining 39 instructions as features for creating the training dataset. The total number of executed instructions for the training software set ranges from 1.8×10^5 to 7.8×10^7. Whisper [4] was used as the functional simulator to execute the ELF files. Verilator 4.028 was used to obtain cycle accurate RTL simulation of the four cores. PMs were then generated by using four different ML algorithms for each RISC-V core. In the prediction phase, we selected 10 benchmarks from TACLeBench [3] that were completely different from the training software set. The benchmarks cover different domains, such as signal processing and mathematical problem solving, and are freely available and designed specifically for embedded systems. After compilation, Whisper [4] was used as the functional simulator to execute the ELF files.

TABLE II
Search Space for ANN Hyperparameter Tuning.

Parameters	Search space
Learning rate	0.001 to 0.2
# hidden layers	0 to 8
# neurons in	8 to 512
Activation in each hidden and output layer	sigmoid/softsign/tanh/ selu/elu/exponential/ LeakyReLU/relu/softplus
Epochs	1 to 1800

TABLE III
MAPE of Cycle Prediction using Different ML Algorithms.

MAPE(%) Core / ML	Ibex	RSD	SweRV	μRV32RTL
OLS Regression	2.2	✗	✗	1.4
LR with MGD	1.8	19.5	2.8	0.4
Ridge Regression	2.1	✗	✗	1.2
ANN	1.6	18.6	6.5	0.4

The application of the ML algorithms and PMs was programmed using Python 3.8. The algorithms were implemented with publicly available libraries, where the OLS regression and ridge regression were implemented using Scikit-learn 1.0 [21] and TensorFlow 2.6.0 [22] was used to implement LR with MGD and ANN. The time required to generate each PM during the training phase ranges from 3 to 1726 seconds, with the time difference based on the complexity of the ML algorithm.

For ANN, we defined a comprehensive search space for hyperparameters and employed a random search [23] technique as the tuner to select the hyperparameters. In each trial, the hyperparameters were randomly chosen from the specified search space. Table II provides a summary of the parameters and the corresponding search space utilized in our hyperparameter tuning approach.

B. MAPE Analysis

We predicted the number of execution cycles for each benchmark on different cores using different ML algorithms. Subsequently, we calculated the MAPE for each core while employing different ML algorithms. The corresponding results are presented in Table III. A MAPE greater than 50% is considered to indicate very low accuracy, to the extent that the prediction

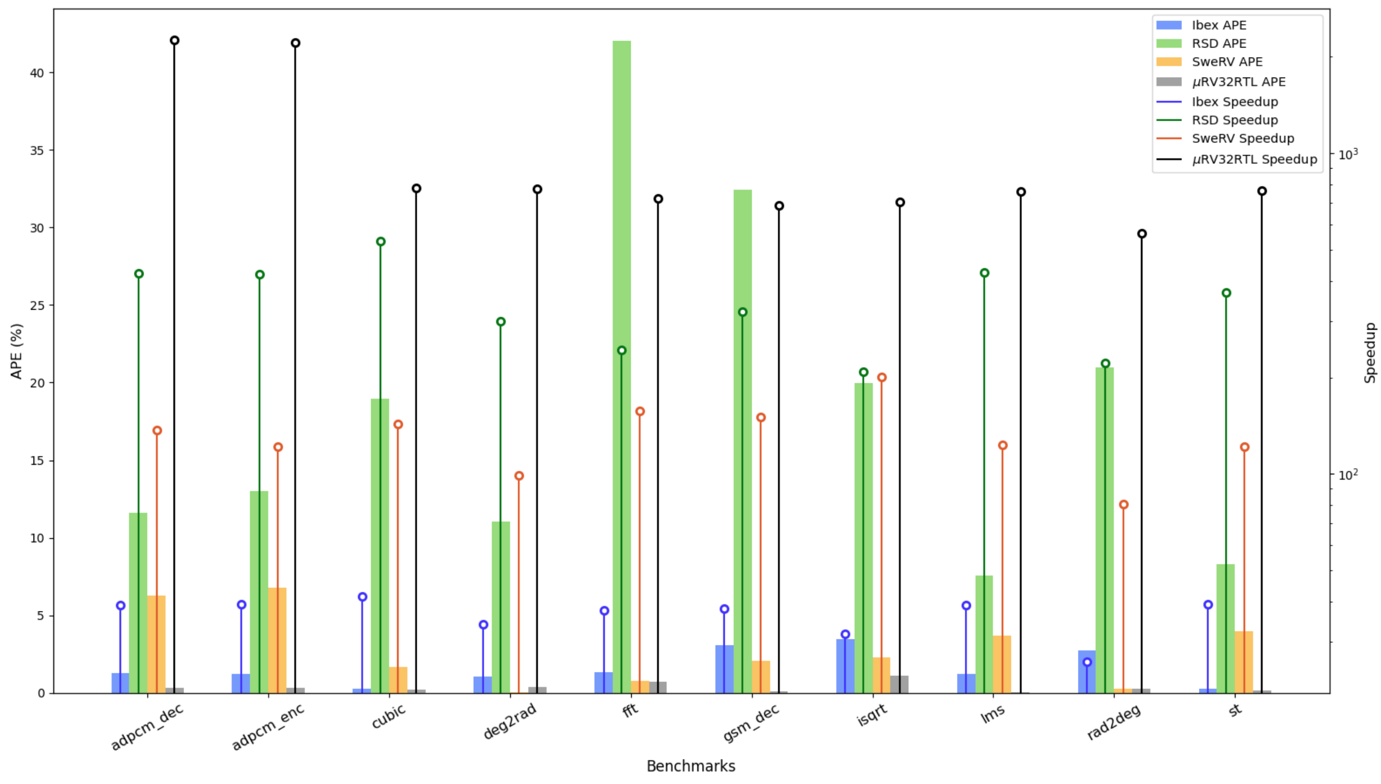

Fig. 3. The APE of the model based on different cores for each benchmark and the corresponding speedup of our approach when predicting the number of cycles.

is not deemed acceptable. In such cases, the low accuracy is visually represented by ✗. For example, OLS regression cannot accurately predict the number of execution cycles on RSD and SweRV. For Ibex, the ANN-based model does not have any hidden layers, and the activation function is LeakyReLU, which can be regarded as linear when the negative half-axis is not considered. OLS regression slightly underperforms the other algorithms, but the difference in performance between the four ML algorithms is not significant. This means that the relationship between the number of execution cycles and the instruction counts can be expressed linearly. For RSD, OLS regression and ridge regression cannot predict the number of cycles correctly, while LR with MGD and ANN have ability of prediction. This may be due to the numerical instability of the training sub-dataset caused by the weak correlation between the number of execution cycles and instruction counts. GD updates coefficients iteratively to solve the problem better. For SweRV, it has the same problem as RSD. For μRV32RTL, it also does not contain hidden layer and the activation function is ReLU, results for different algorithms can be seen as equivalent. Therefore, PM for μRV32RTL can be expressed by the linear equation.

When using our approach to predict microarchitectural behaviors for Ibex and RSD, we found that that our approach yields accurate results for Ibex but is ineffective for RSD. The reason can be that other microarchitectural behaviors are not related to instruction counts or considering only instruction counts is insufficient in capturing the microarchitectural behaviors. The PM for Ibex achieves a MAPE of 1.3% for LSU Busy counter and 7.3% for the Fetch Wait counter using ridge regression.

C. APE and Simulation Time Analysis

For each core, we chose the best PM by finding the corresponding ML algorithm with the smallest MAPE. At the same time, we compared the speedup between our PM and the corresponding RTL core, while accounting for the prediction phase simulation time, which is influenced by two primary factors: the utilization of a functional simulator to obtain runtime information of embedded software, and the use of PM for performance prediction. However, due to the negligible time spent on PM compared to the functional simulator, the simulation time incurred by the functional simulator can be considered as the simulation time for new software.

Fig. 3 illustrates the APE for each benchmark from the best PMs and the speedup for each benchmark when predicting the number of cycles. The x-axis represents 10 benchmarks, while the left y-axis uses a linear scale to show the results for APE in the form of bars. On the other hand, the right y-axis is logarithmically scaled to represent the results of speedup, with marked vertical lines. The logarithmic scale is chosen for the right y-axis due to its wide dynamic range and ability to visualize exponential relationships. For Ibex, the PM based on the ANN is selected and the APE of each benchmark is less than 3.5%, which means that this PM is capable of modeling Ibex core. Our approach achieves up to $41.4\times$ faster simulation speed than RTL simulation using Ibex. PM for RSD is based on the ANN and the corresponding APE ranged from 7.5% to 42.0%. The underlying reason is that instruction counts are insufficient to represent the full features related to the number of cycles for RSD. For simulation speed, our approach is $208.4\times$ to $531.2\times$ faster than RSD. For SweRV, the best PM is based on the LR with MGD. The APE ranges from 0.04% to 6.8%, which means that PM can roughly mimic the execution cycle behavior of the SweRV core. Our approach can simulate $80.4\times$ to $200.6\times$ faster than SweRV. The best PM for μRV32RTL is based on LR with MGD with a maximum APE of 1.1%, which means this PM can accurately model the μRV32RTL core in a linear fashion. Compared to μRV32RTL, our approach achieves a speedup of $563.3\times$ to $2261.4\times$ in simulation speed. μRV32RTL simulation is very slow compared to other cores. Therefore, for each benchmark, the PM for μRV32RTL always achieves the maximum speedup. It is meaningful to use our approach to estimate the number of execution cycles on μRV32RTL.

V. Conclusion and Future work

In this paper, we present a novel ML-based approach for estimating the performance of embedded software on RISC-V processors, and we demonstrate its generalizability across four different microarchitectures. Our approach estimates the performance of RISC-V processors by combining a fast functional simulator and ML-based models. In our experiments, four ML algorithms were implemented for each RTL core, and the best PM was determined after comparing the MAPE generated using different ML algorithms. For each RTL core, our approach is able to predict the number of execution cycles quickly and accurately.

In the future, we plan to extend our approach towards more performance-related features which are not dependent on instruction counts. Additionally, we plan to extend the approach to multi-core processors or other computer architectures.

References

[1] N. Wu and Y. Xie, "A survey of machine learning for computer architecture and systems," *ACM Computing Surveys (CSUR)*, vol. 55, no. 3, pp. 1–39, 2022.

[2] W. Zhang, M. Goli, A. Mahzoon, and R. Drechsler, "ANN-based performance estimation of embedded software for risc-v processors," in *33rd International Workshop on Rapid System Prototyping (RSP)*, 2022.

[3] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *International Workshop on Worst-Case Execution Time Analysis (WCET)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[4] "Whisper ," https://github.com/chipsalliance/VeeR-ISS.

[5] M. Goli and R. Drechsler, "Automated design understanding of systemc-based virtual prototypes: Data extraction, analysis and visualization," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2020, pp. 188–193.

[6] J. Silveira, L. Castro, V. Araújo, R. Zeli, D. Lazari, M. Guedes, R. Azevedo, and L. Wanner, "Prof5: A risc-v profiler tool," in *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2022, pp. 201–210.

[7] M. Lattuada and F. Ferrandi, "Performance estimation of embedded software with confidence levels," in *Asia and South Pacific Design Automation Conference*. IEEE, 2012, pp. 573–578.

[8] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2015, pp. 52–59.

[9] W. Zhang, M. Goli, and R. Drechsler, "Early performance estimation of embedded software on risc-v processor using linear regression," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2022, pp. 20–25.

[10] V. Muttillo, P. Giammatteo, and V. Stoico, "Statement-level timing estimation for embedded system design using machine learning techniques," in *ACM/SPEC International Conference on Performance Engineering*, 2021, pp. 257–264.

[11] M. S. Oyamada, F. Zschornack, and F. R. Wagner, "Applying neural networks to performance estimation of embedded software," *Journal of Systems Architecture*, vol. 54, no. 1-2, pp. 224–240, 2008.

[12] I. Hafnaoui, R. Ayari, G. Nicolescu, and G. Beltrame, "A simulation-based model generator for software performance estimation," in *Summer Computer Simulation Conference*, 2016, pp. 1–8.

[13] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 3, pp. 257–279, 1999.

[14] "Ibex RISC-V Core," https://github.com/lowRISC/ibex.

[15] "RSD RISC-V Core," https://github.com/rsd-devel/rsd.

[16] "SweRV RISC-V Core," https://github.com/chipsalliance/Cores-VeeR-EH1/tree/1.0.

[17] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "The microrv32 framework: An accessible and configurable open source risc-v cross-level platform for education and research," *Journal of Systems Architecture*, vol. 133, p. 102757, 2022.

[18] "RISC-V GNU Compiler Toolchain," https://github.com/riscv-collab/riscv-gnu-toolchain.

[19] W. Snyder, "Verilator," https://www.veripool.org/wiki/verilator.

[20] "SpinalHDL," https://github.com/SpinalHDL/SpinalHDL.

[21] "Scikit-learn," https://scikit-learn.org/stable/.

[22] "TensorFlow," https://www.tensorflow.org.

[23] "Random search," https://keras.io/api/keras_tuner/tuners/random/.