CATREEN: Context-Aware Code Timing Estimation with Stacked Recurrent Networks

Abderaouf Nassim Amalou, Élisa Fromont, Isabelle Puaut

CATREEN: Context-Aware Code Timing Estimation with Stacked Recurrent Networks

Abderaouf N Amalou
Univ. Rennes, INRIA, CNRS, IRISA
Rennes, France
abderaouf.amalou@irisa.fr

Elisa Fromont
Univ. Rennes, IUF, INRIA, CNRS, IRISA
Rennes, France
elisa.fromont@irisa.fr

Isabelle Puaut
Univ. Rennes, INRIA, CNRS, IRISA
Rennes, France
isabelle.puaut@irisa.fr

***Abstract*—Automatic prediction of the execution time of programs for a given architecture is crucial, both for performance analysis in general and for compiler designers in particular. In this paper, we present CATREEN, a recurrent neural network able to predict the steady-state execution time of each basic block in a program. Contrarily to other models, CATREEN can take into account the *execution context* formed by the previously executed basic blocks which allows accounting for the processor micro-architecture without explicit modeling of micro-architectural elements (caches, pipelines, branch predictors, etc.). The evaluations conducted with synthetic programs and real ones (Programs from Mibench and Polybench) show that CATREEN can provide accurate prediction for execution time with 11.4% and 16.5% error on average, respectively and that we got an improvement of 18% and 27.6% respectively when comparing our tool estimations to the state-of-the-art LSTM-based model.**

I. Introduction

The complexity of developing cycle-accurate simulators and integrating them into compiler infrastructures has led compiler designers to use much simpler ways to estimate execution times to decide about the compiler optimizations to apply, the simplest ones being architecture-independent cost functions, e.g., simply counting the number of machine instructions. An alternative, reachable thanks to the recent advances in Machine Learning (ML), consists in predicting the execution times of code snippets [1] to guide optimizations [2]. The benefit of using ML for timing prediction is threefold: (i) no detail of the processor microarchitecture is needed because the behavior is learned from timing measurements; (ii) porting the ML-based execution timing predictor to a new microarchitecture does not need any deep expertise, a new training step is only required; (iii) even if training an ML model is a time-consuming task, prediction is in general fast, allowing ML-based timing predictions to be used in compilers.

Today's processors feature increasing complexity (e.g., cache hierarchy, pipelines, branch predictors, instruction scheduling in out-of-order cores), which, combined with the lack of associated documentation, makes building a cycle-accurate simulator like [3] for each new architecture time-consuming and error-prone. This complexity is mainly due to the integration of several hardware accelerators, which aim to speed up the execution time of programs. The cohabitation of these components makes the timing modeling of a processor difficult and sometimes unpredictable. Let us take, for example, a simple processor architecture, that has an N-stage pipeline, data and instruction caches, and a branch predictor. The **pipeline** allows the execution of a new instruction to start without waiting for the previous one to finish as long as there are no register dependencies. The **cache memory** is a small, fast memory that stores copies of instructions/data from frequently used locations in the main memory. And finally, the **branch predictor** allows the processor to speculate about the outcome of a conditional branch (e.g., fill in the pipeline with the instructions of the predicted branch direction) to accelerate the execution.

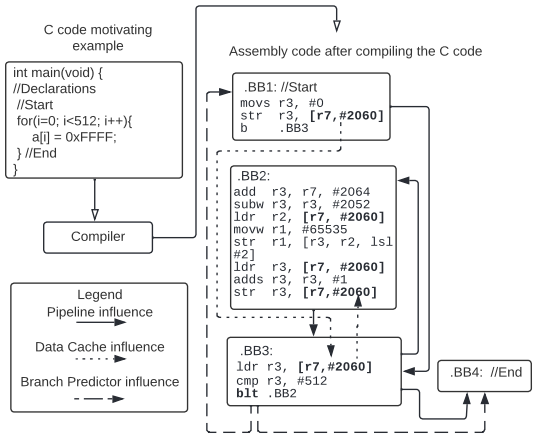

Fig. 1: Inter Basic Blocks hardware dependencies.

These components introduce dependencies between successive instructions, making the timing of a sequence of instructions dependent on its *execution context*. This is illustrated in Figure 1, the left top square represents a C code snippet delimited by two *comments* //Start and //End and the right boxes represent the machine instructions after compilation, separated into *basic blocks*[1] (BB). The binary code in our example is made of 4 BBs denoted in Figure 1 as BB1, BB2, BB3 and BB4.

Regarding the effect of **pipelines**, the execution time of a BB depends on the BB executed before, which defines the level of parallelism between the successive BBs. For example, the execution time of BB3 must take into account that BB2

[1]A basic block (BB) is a sequence of instructions that has a single entry point and a single exit point

has been executed before. Plain arrows in Figure 1 show all pairs of BBs concerned by this effect. As far as the **branch prediction** is concerned, if the branch predictor successfully predicts the outcome of the branch instruction contained in BB3 (the "blt" instruction, which means: branch if i is lower than 512), the first instructions of BB2 are executed before the result of the comparison is known. This makes the timing of BB2 dependent on the previous execution of BB3. Similarly, in the case of an incorrect branch prediction, the timing of BB4 depends on BB3. These effects are depicted by dashed arrows. Regarding the effect of **caches** on our example, if the variable i is loaded in the cache in basic block BB1 and is not evicted from the cache when BB2 is executed, the execution time of BB3 is reduced, making the timing of BB3 dependent on the previous execution of BB1 and BB2. These cache-related effects are depicted by dotted arrows in Figure 1.

In this paper, we introduce CATREEN: Context-Aware code Timing estimation with stacked REcurrEnt Networks. CATREEN infers the steady-state execution times of individual BBs in programs (at the assembly code level). As compared to related works on estimating the execution time of BBs, the novelty of our work is to assess the execution time of a BB within its *execution context*. Instead of calculating the best-case execution time of a basic block b considered in isolation as in the closest related work, ITHEMAL [4], CATREEN calculates the execution time of b after executing a sequence of other BBs.

TABLE I: The importance and influence of the context size on the error made (the lower the better) when estimating the execution time of the program in Figure 1, using CATREEN on a hardware platform that contains a pipeline, a cache, and a branch predictor.

Context size (Nb. of BB)	0	1	5	10	20	100
error on average	22%	22%	19%	17.5%	17.5%	18%

Table 1 motivates the utility of context-awareness to estimate the execution time, when using CATREEN on hardware and the program described above (in Figure 1), we observe that the errors made tend to decrease with the increase of the context size (number of basic blocks that we consider to form the context), we also found that error-value saturates around 18% for this program. This example explains our interest in context-aware timing estimation which is more faithful to the real execution time of a BB, as it captures timing effects resulting from hardware-software interactions in execution sequences larger than a BB. Moreover, experiments have shown that our proposal produces more accurate predictions than state-of-the-art context-agnostic ML-based techniques. Additionally, contrary to most of the previous work on timing models, our paper targets specifically embedded architectures for the following reasons: (i) embedded processors are performance-sensitive (for energy and safety reasons), hence the need for a precise performance model. (ii) today's embedded platforms are sufficiency complex to make the use of simulators cumbersome. (iii) in the embedded development process, access to the physical hardware platform may not be granted, so acquiring early insight into the performance is crucial to choose the best processor for the developed application.

The paper is organized as follows. Section II surveys related works that use ML-based techniques for performance evaluation and optimization. An overview of CATREEN is given in Section III. The performance of CATREEN is evaluated in Section IV. Finally, Section V concludes with a summary of the contributions and directions for future work.

II. Related work

The complexity of evaluating and improving the performance of programs has motivated the use of machine learning techniques. These techniques can be classified into two categories: those which directly evaluate the execution time of programs (§ II-A) and those which target other metrics that are related to performance like speedup or energy consumption, albeit not directly evaluating execution times (§ II-B).

A. Machine learning for execution time prediction

ITHEMAL [4] leverages a hierarchical multi-scale Recurrent Neural Network (RNN) and, in particular, Long Short-Term Memory (LSTM) layers to predict the throughput of basic blocks. ITHEMAL captures the interactions between instructions from the same basic block. Each basic block is isolated from the program using a dynamic binary instrumentation tool [5] and executed repetitively to reach its steady-state behavior and obtain its throughput (peak performance or *best-case* execution time). In contrast to ITHEMAL, CATREEN relaxes the assumption that a basic block is executed in isolation and predicts the execution time of a basic block given its actual context of execution at steady-state.

[6] advocates the use of sparse polynomial regression to predict the execution time of programs given a set of static features. CATREEN differs from [6] by the features used during learning that allow capturing the execution context of basic blocks. It also differs by the type of method used to predict the throughput (in our case, a recurrent neural network).

Seshia and Rakhlin [7] propose GameTime, a game theory approach to predict the distribution of program execution times. They formulate the problem as a player (the predictor) that seeks to win a game (accurately predict time statistics for a program) versus an adversary (the program's environment). GameTime, contrary to our work, executes the program. Meanwhile, CATREEN predicts the execution time without executing the code.

Other than Machine Learning techniques, Ritter and Hack [8] present a framework for inferring port usage of instructions based on an evolutionary algorithm that solves a linear program to predict the throughput of a basic block. the solution was designed to infer port mappings for the out-of-order processor but it is not clear how the approach could be extended to infer the throughput for a more complex trace execution, e.g., how data dependencies could be included in the analysis? In comparison, CATREEN was designed for in-order processor

architecture due to the sequential processing of instructions, but it has the ability to learn instruction and data dependencies.

B. Machine learning for performance optimization

ML techniques were proposed in the past to improve the performance of programs [1], [9]–[12].

The DeepTune [9] tool leverages an RNN LSTM-based model that can be used as a classifier to choose the best choice between running a program on the CPU or the GPU, or to predict the best number of threads among a set of values. In [10], a neural network (NN) model uses performance counters to predict the performance (instructions per cycle) of a thread migration from one core to another in S-NUCA architecture. [12] uses static program features to predict the number of threads and the scheduling policy for an efficient task mapping to multi-core processors using neural networks. Tousi and Lujan [1] study the feasibility of using classic Machine learning techniques to predict performances (latency, speedup...) on the SPEC Benchmarks, while CATREEN only uses programs that can run without an operating system (bare-metal environment to avoid OS noises) which is not the case of SPEC Benchmarks. Performance counters are also employed in [11] to determine good compiler optimization settings through the training of a logistic regression model. If the methods presented here and their inputs are (somewhat) similar to ours, our end goal is entirely different. CATREEN, in contrast to the works mentioned above, focuses on the prediction of execution time for a given hardware and compiler settings.

III. Execution time prediction using CATREEN

CATREEN leverages a stacked recurrent neural network architecture to predict the execution time of a basic block b (or the *basic block under analysis*) given its execution context (sequence of basic blocks executed before b). By considering the execution history of basic blocks, CATREEN naturally accounts for the state of the hardware when executing the basic block (pipeline, cache hierarchy, and branch prediction).

A. CATREEN Overview

Recurrent Neural Networks (RNN) are particular neural network architectures that can be trained to predict outputs from a variable-length sequence of data. These networks can memorize (parts of) the sequence and the computed sequential information along time. Long Short-Term Memory (LSTM) are special kinds of RNN, capable of learning long-term dependencies [13] in the sequences. LSTM architectures have mechanisms, called gates, that are trained to choose whether or not to keep particular sequential information. These architectures are, for example, extensively used in domains such as natural language processing [14] [15], where the context of a word in a sentence is useful to learn its characteristics. Figure 2 represents the overall architecture of CATREEN. CATREEN estimates the execution time of a basic block within a sequence of basic blocks in a similar way that an LSTM would process a paragraph in natural language to predict, for example, the sentiment (positive or negative) of a given sentence in this paragraph. The analogy is as follows: an instruction represents a word. An instruction is composed of an opcode and one or more operands, which can be treated as letters that constitute this word. A set of instructions will form a basic block (a sentence), and a sequence of basic blocks will therefore represent a paragraph.

The architecture of CATREEN is composed of five layers, depicted in Figure 2 from top to bottom. The first layer is a *tokenization and embedding* layer that pre-processes the assembly language, extracted from the machine code, and generates inputs that are understandable by the next layers. The next three layers are the central layers of CATREEN. They are LSTM layers (called RNN in the following): one to process an instruction (Instruction Layer in Figure 2), one to process a basic block (BB layer in Figure 2), and one to process a sequence of basic blocks (Sequence Layer in the Figure 2). Finally, we create a sort of weak connection to save the basic block under analysis (last BB in the sequence) and concatenate it with the formed context and send it to a dense layer that is used to predict the final timing output (timing a basic block in the context of the previously executed basic blocks). More details on each layer are given below.

B. Tokenization and embedding

In order for the RNN to properly interpret basic blocks, an encoding of machine code into a sequence of integers is needed, where each integer represents an index (token) in a predefined dictionary. This is performed as follows. We pre-process the raw data (assembly code) by encoding each basic element of an instruction (opcode and operand) with a unique number (token), with a special treatment given to the operands:

- All immediate operands are encoded with the same token.
- All addresses are encoded with the same token.
- Addressing modes using registers are differentiated by assigning a distinct token value per pair (register number, addressing mode) with the addressing modes supported by the considered architecture (e.g., for the ARM targets, direct access, indirect access, and indirect access with offset).

For example, consider the following sequence of ARM instructions: *MOV R3, #0x0 ; LDR R3, [R3] ; BL 0x080008DC ; STR R3, [R3, #0x0166]; MOV R3, #0x0230*.

The final output of the encoding for the considered sequence of instructions is:*[19, 500, 380, 999, 28, 500, 501, 999, 65, 381, 999, 35, 500, 502, 999, 19, 500, 999]*. Where 999 is a separator between instructions.

In order to be suitable inputs for RNN layers, each token is again encoded into a distinct fixed-size vector of floats within the interval [-1,1] using word2vec [16].

C. LSTM RNN layers of CATREEN

Since we do not, *a priori*, know what is the length of the sequences (i.e., how many basic blocks they contain) nor the basic blocks' length (i.e., how many instructions they contain) nor even the length of the instructions (i.e., how many operands they contain), we model these data with LSTM, which are

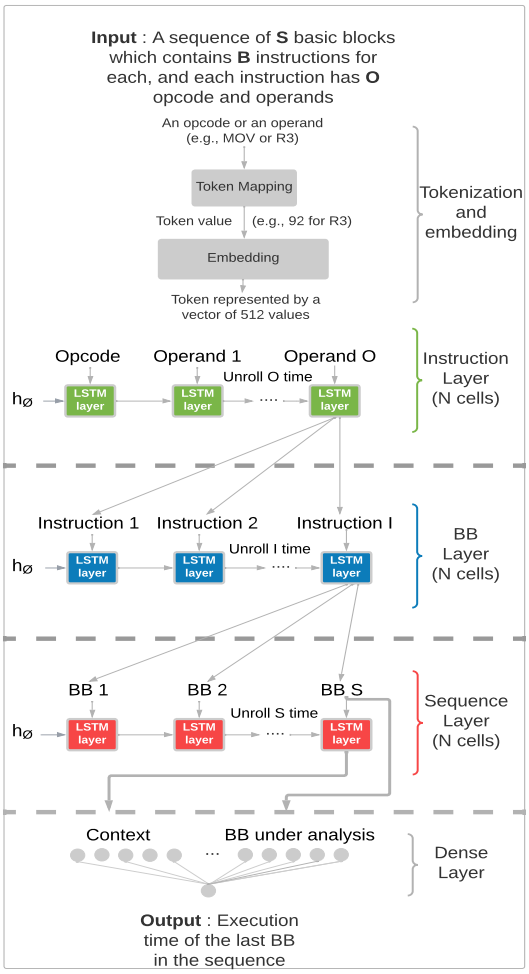

Fig. 2: Architecture of CATREEN. The input is a sequence of basic blocks consisting of a sequence of instructions which are themselves, sequences of operands/opcodes. CATREEN regresses a timing estimation for the last basic block in the input sequence.

suited to variable-length sequences. Three levels of LSTM are used for that purpose:

- The *Instruction Layer* (in green in Figure 2) is the first LSTM layer in CATREEN. It takes as input a sequence of embedded code operands/opcodes (i.e., an instruction). We loosely denote the length of the sequence of operands/opcodes by O (resp. I and B in the subsequent layers) even though it differs from one instruction to another. This LSTM layer processes the entire sequence of embedded tokens from an instruction (indexed by $t \in [0..O-1]$) and produces a single final output representation at $t = O - 1$ for the entire instruction (please refer to [13] for more information about the inner working of an LSTM). To avoid direct influence between instructions, the hidden state is reset to its initial state $h\emptyset$ at each instruction (resp. at each BB and sequence), this helps to learn more longer sequence where each representation is sent to the next stage to be better interpreted: the outputs (instructions) are treated one by one by the next LSTM layer (BB layer, for Basic Block layer). We loosely denote the dimension per layer by N even though it may differ for the three RNN layers. The selected hidden size per layer is given in Section IV-D.
- The *BB layer* (in blue in Figure 2) processes the sequence (of length I) of embedded instructions in a basic block and produces a representation for each basic block. Again, all representations of a basic block are treated one element by one by the next LSTM layer (Sequence layer) with the reset of the hidden state.
- Finally, the *Sequence Layer* (in red in Figure 2) processes the sequence of S basic blocks within a sequence. Similarly to the other RNN layers, it produces at the end a representation of the sequence or what we call *Context*. Then, the value of *Context* is concatenated to the value of the *BB under analysis* (that we save from the previous layer *BBs*) and they are given as inputs to a fully connected linear layer connected to a single output which produces an estimate of the execution time of the last basic block of the sequence given the execution history.

IV. Experimental evaluation

Before evaluating CATREEN, we describe the data used to train and test our tool in Section IV-A, followed by the experimental setup in Section IV-B. In Section IV-C, we present the context-agnostic baselines used to assess our method. Then, we show that a careful selection of hyper-parameters is crucial for high precision results, for both CATREEN and the baseline predictors (Section IV-D). We then demonstrate that CATREEN outperforms the baseline context agnostic techniques, on both synthetic code (Section IV-E) and real code (MiBench2 and PolyBench programs, Section IV-F). Section IV-G finally presents and compares the inference time of the different techniques.

A. Training and test data

The supervised training of our stacked LSTM network requires a large number of labeled data samples. In order to cover a large number of possible programs, we use synthetic codes. To test our model, both synthetic and real programs are used. In both cases (training and testing), we use a hardware-assisted solution to obtain a timed execution trace from these programs (*ground truth* timing data used as label for supervised learning). The execution trace[2] is then pre-processed to constitute suitable inputs for our model.

1) Synthetic data: A code generator was developed to produce varied C source code programs. The code generator produces programs that randomly manipulate a selected number of statements and variables (the user provides parameters specifying the proportion of each element from a declared grammar). The code generator produces source code with loops, if-then-else constructs, and uses all the elementary types from the C language (*integers*, *floats*, etc.). Basic statements

[2]Instruction-by-instruction representation of the program execution.

use the most common operations available in C (arithmetic and logical operations, shift and rotate operations, binary and unary operators or booleans, etc.). Arrays are also covered with various access modes: constant (access to a constant index), sequential, linear (affine loop indices), and random. The code generation ensures the absence of run-time errors by construction (out-of-bound array accesses, divide by zero, etc.).

2) Real Data: We evaluate our approach on a dataset of real programs, MiBench 2[3], a benchmark suite based on MiBench [17] and ported for IoT devices. Our tests were carried out on the *Automotive and Industrial Control* category of MiBench. We also use PolyBench v4.2 [18] a set of programs that has the specificity to manipulate nested loops.

3) Ground truth timing generation: Training and validating CATREEN have to be performed with accurate timing values. Moreover, the way the timing values are obtained should not change the timing of the code under execution (well-known as *probe-effect*). In CATREEN, we opted for a hardware-based solution, using the Joint Test Action Group (JTAG) interface. The J-Trace Pro trace solution from Segger [19] is used to connect to the JTAG interface of the target processor, alongside Ozone [20], a cross-platform debugger and performance analyzer. Ozone generates execution traces with a format of one line per machine instruction. Each line contains information about the value of the cycle counter after executing the instruction, the address of the instruction, its opcode and operands, and the corresponding assembly code.

4) Data pre-processing: To create the final data (both for synthetic and real programs), the execution trace generated by Ozone is processed as follows. First, basic blocks (BBs) are extracted and the execution time of each BB is calculated. In order to eliminate outliers, the ground truth timing of each BB is obtained by using multiple executions and keeping the median value. CATREEN is trained by using a normalized timing value equal to the median value of the measurements divided by the number of instructions in the BB. Then, the opcodes and operands of instructions are tokenized as explained in Section III-B from a dictionary built using the ARM armv7-m manuals [21]. Finally, sequences of BBs are formed, in which the prefix sequence of a BB (up to the last one) corresponds to the context, and the last BB is the block under timing estimation. For example, if we have a sequence [BB0, BB1, BB2, BB3], BB3 is the basic block under analysis, and the sequence [BB0, BB1, BB2] corresponds to the context of BB3. Sequences that are too short to have enough contextual information (in our experiments, sequences shorter than 5 BBs) are filtered out. All other generated sequences are included in the data, regardless of their length.

5) Representativity of synthetic programs w.r.t. real programs: Our synthetic dataset should be as representative as possible of real code. To investigate this, we compare the proportions of 4 typical classes of instructions (move, branches, load/store and arithmetic and logic instructions) on three sets of programs: (i) a set that contains 1000 synthetic programs produced by our generator; (ii) the MiBench automotive programs, and (iii) the AnghaBench benchmark suite, that contains 1 million compilable C programs, albeit **non-executable** [22]. Figure 3 depicts the results as box-plots; it shows that for each benchmark, the medians of a given category of instructions are quite close. Synthetic programs are short programs, which explain the lower median of the mov and load/store instructions. Regarding the variability, synthetic data and AnghaBench have more variability than MiBench because of the small size of this latter benchmark (5 specialized programs). The synthetic programs and AnghaBench, on the other hand, covers a large set of C programs, which implies a wider range of proportions.

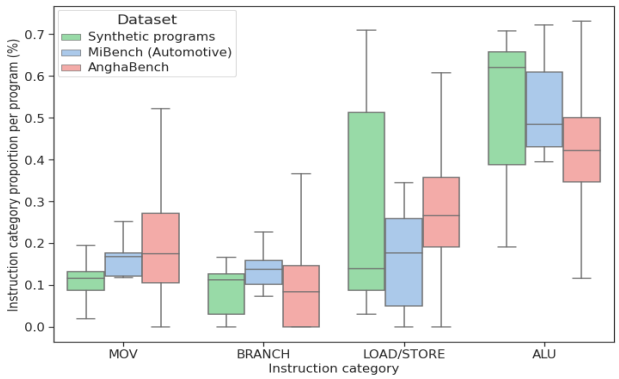

Fig. 3: Variation of machine instructions categories proportion per program on synthetic code, MiBench Automotive and AnghaBench.

B. Experimental setup

Experiments were performed on an STM32H7 board. The board features an ARM Cortex M7 processor, which has a 6-stage in-order pipeline, 16 KB instruction and data caches, and a branch predictor. We generated 1000 synthetic code snippets that we execute on the bare-metal target processor to eliminate any possible interference and extract traces using the JTAG interface. After the pre-processing data phase to get sequences (as explained in Section IV-A4), 10000 different sequences were available for training, 2000 different sequences for validation, and 1000 different sequences for testing. Each subset of data (training, validation, test) comes from a separate set of programs to remove the bias it would otherwise introduce. We performed 1000 execution time measurements for each basic block, with cache warming before sampling (cache warming is performed by executing the program 20 times). All learning models were implemented in PyTorch [23] and were trained on NVIDIA GeForce RTX 2080 Ti. CATREEN training lasted four days for each setting (set of values of hyperparameters).

C. Baseline ML-based execution time predictors

The performance of CATREEN is compared with two context-agnostic execution time predictors. The most simple, albeit not naive one, is a Multi-Layer Perceptron regressor

[3]https://github.com/impedimentToProgress/MiBench2

(denoted as *MLPr* in the following). An MLPr is a feed-forward neural network that does not take into account sequential information and requires a fixed-size input. Such a baseline has been successfully used for timing prediction in [24]. Specifically, the MLPr we have implemented takes as an input 233 static features of the basic blocks: the proportions of each type of machine instruction (e.g., MOV, ADD, LDR); the proportion of the different addressing modes (e.g., immediate, direct access, register indirect access, register indirect access with offset). We used a grid search algorithm to select proper MLPr hyperparameters (number of hidden layers, optimizer, learning rate, and the loss function). The result of this search gave us the following ideal (on the validation dataset) parameters: {hidden_layer_sizes=(256, 256), learning_rate='adaptive', learning_rate_init=0.001, solver='adam', loss_function='mean squared error'.}. Our second baseline is ITHEMAL [4], that similarly to CATREEN, uses RNNs for execution time prediction. More precisely, we compare CATREEN with two versions of ITHEMAL. The first one is a direct re-implementation of ITHEMAL from the original paper (denoted as *ITHEMAL-untuned* in the following), using the hyperparameters suggested by the authors. The re-implementation consisted in porting the tokenization/embedding step of ITHEMAL to the ARM instruction set and using a GPU for training instead of a parallel CPU. For the second version of ITHEMAL (called *ITHEMAL-tuned* hereafter), we have tuned the hyperparameters of the model to better fit the new data.

D. Hyperparameters tuning

As often discussed in research papers using recurrent neural networks (e.g., [25], [26]), tuning hyperparameters is of utmost importance to guarantee the training convergence and the generalization performance of the learned models on new data. However, this tuning phase is computationally expensive and thus, only a limited number of configurations can be tested. We show in the following how we have tuned (on validation data) our learning hyperparameters and how this can drastically improve the performance (in terms of Mean Absolute Percent Error, MAPE [27]) of both CATREEN and our closest competitor, ITHEMAL. To reduce the hyperparameter exploration space, we did not tune the *architecture* of the models since we started with ITHEMAL as a base architecture. We selected for CATREEN the same main architectural parameters as initially selected for ITHEMAL: each LSTM layer contains 256 LSTM cells, so the output of the LSTM layer is a 256 dimensional vector. We did not vary the number of training epochs (20) since this was the best we could do with our computation power. After preliminary experiments, we have changed the token embedding size from 256 (as in the original ITHEMAL) to 512. This is due to our encoding of ARM instructions, which requires more tokens than the encoding used by ITHEMAL. We have studied the impact of three learning hyperparameters:

- The *optimization algorithm*, by comparing two optimizers: Stochastic Gradient Descent (SGD), used in ITHEMAL [4] and ADAM optimizer [28] widely used in deep learning.
- The *learning rate*. ITHEMAL uses an adaptable learning rate with an initial value of 10^{-1} which decreases by a factor of 1.2 every epoch after the first two epochs. We consider that 10^{-1} is a high learning rate, so we chose to explore lower values. However, a low learning rate considerably increases the training time. Thus, we have explored only three learning rates: two constant values 10^{-3} and 10^{-4}, and an adaptive that starts from 10^{-2} and decreases at each epoch by a factor of 10 until 10^{-4}).
- The *loss function*, by comparing two regression losses: the one used in ITHEMAL (Mean Absolute Percentage Error) and the symmetric Mean Absolute Percentage Error loss function [27] which is neutral regarding under or over-forecasting and seemed more appropriate:

$$MAPE_{loss} = \frac{1}{n} * \sum_{i=0}^{n} \frac{|predict_i - actual_i|}{actual_i} \quad (1)$$

$$sMAPE_{loss} = \frac{2}{n} * \sum_{i=0}^{n} \frac{|predict_i - actual_i|}{predict_i + actual_i} \quad (2)$$

The sensitivity of CATREEN and ITHEMAL to hyperparameter tuning is shown in Table II. Several observations can be made from the results. First, we observe that ADAM (which was not used in [4]) drastically reduces the generalization error of the learned models. Second, regarding the learning rate, the best results are obtained in all cases with a learning rate of 10^{-4} except when sMAPE+SGD is used, where 10^{-3} gives the best results (12% and 20%) both from CATREEN and ITHEMAL. The adaptive version (adapt) gives the worst results in most of the tests (except for ITHEMAL, when MAPE+SGD is used, in this case, it is the constant value 10^{-3} that gives the worst error percentage). Finally, we observe that sMAPE generally gives better results whatever the other parameters. MAPE is asymmetric by definition and puts a heavier penalty on over-estimation errors (when predictions are higher than ground truth) than on under-estimation errors. As a result, MAPE will favor models that are under-forecast. Overall, the results reported in Table II show that the hyperparameter selection has a huge impact on the model performance (i.e., the error estimated on the validation set). They further show that the hyperparameter selection, as done in the original version of ITHEMAL, is sub-optimal for our data. In the next sections, we use CATREEN and ITHEMAL with their best-found hyperparameters (sMAPE, ADAM, 10^{-4}), and similarly for ITHEMAL (MAPE, ADAM, 10^{-4}).

E. Prediction performance on synthetic data

We compare the performance of MLPr, *ITHEMAL-untuned* [4], *ITHEMAL-tuned* and CATREEN on the 1000 test sequences. The results are reported in Table III and they are given as a MAPE, where the mean is computed over the 1000 samples and the MAPE is expressed as a percentage. We also provide a Spearman score (rank correlation score) and Pearson score (linear correlation score) test for each method. The best results in the table are highlighted in bold. We observe that CATREEN gives the best results, with a MAPE of 11.4% an

TABLE II: MAPE performance of CATREEN and ITHEMAL, for different learning hyperparameters (loss function, optimizer, learning rate)

Loss function	MAPE						sMAPE					
Optimizer	SGD			ADAM			SGD			ADAM		
Learning Rate	10^{-3}	10^{-4}	adapt	10^{-3}	10^{-4}	adapt	10^{-3}	10^{-4}	adapt	10^{-3}	10^{-4}	adapt
CATREEN	**18%**	**18%**	**19%**	**11%**	**13%**	**17%**	**12%**	**16%**	**17%**	**11%**	**10%**	**13%**
ITHEMAL	39%	24%	38%	18%	17%	21%	20%	27%	28%	25%	25%	25%

improvement of 18% compared to ITHEMAL tuned (which was trained on the same data set for a fair comparison). This means that the model has benefited from the execution history of a given basic block to predict its timing, as further detailed below. In comparison, the other techniques that are context-agnostic provide more modest results; the simplest technique (MLPr) gives the worst results, which shows that handcrafted features (even though we introduced a large number of features, here 233) are less discriminant than the learned ones. Our results also show the sensitivity of ITHEMAL to hyperparameters and, therefore, the utmost importance to tune them to improve the results.

TABLE III: MAPE of timing predictions for 1000 basic blocks by four different ML-based methods.

ML technique	MAPE	Spearman	Pearson
MLPr	28.8%	0.974	0.966
ITHEMAL-untuned	27.4%	0.973	0.968
ITHEMAL-tuned	13.9%	0.972	0.975
CATREEN	**11.4%**	**0.983**	**0.977**

F. Timing prediction on benchmarks

We evaluate our approach on a dataset of real programs, the Automotive program set from MiBench 2 and PolyBench. The execution time is estimated on the entire program. For *susan*, different treatment options are possible (*corner+edges* and *smoothing*), so we tried them both individually, for PolyBench programs, we choose to use small dataset option because of the flash memory size limitation for programs when running them on bare-metal (we also had to drop some programs that do not fit on the flash memory). The same experimental process as for the synthetic dataset (use of JTAG/Ozone, see Section IV-A) was used to (i) generate the sequences of basic blocks serving as inputs for the timing predictions, (ii) obtain the ground truth timing values. Table IV reports the Absolute Percentage Error ($APE = 100 * \frac{|prediction - actual|}{actual}$) for the different programs, for CATREEN, ITHEMAL-tuned and MLPr. The results of ITHEMAL-untuned were not good enough to deserve their reporting in the table. The best APE values are highlighted in bold. The results show that CATREEN outperforms ITHEMAL-tuned and MLPr, with a lower APE on all benchmarks (16.5% on average), except for mvt program. In general, we can observe that ITHEMAL-tuned tends to have good results in PolyBench programs which can be explained by the nested loops that constitute them. These loops make the execution time of the BBs steady (in best case), which can hide the context effect on these BBs. The use of a completely context-agnostic technique like MLPr gives, as expected, the worst predictions. The average prediction error of CATREEN on the tested benchmarks is comparable (although slightly higher) to that obtained on the synthetic codes. This shows that our synthetic dataset is globally representative of real code. The slightly higher error may come from two factors (These factors will be more deeply explored in future work): (i) Not sufficiently representative code generation regarding array accesses (only basic array access patterns are produced by our code generator). This can be addressed by improving our dataset, for instance, by synthesizing training data from real code, like done in [29]; (ii) Too compact (therefore imprecise) tokenization of memory accesses (CATREEN does not consider different memory addresses as distinct tokens, in order to avoid an explosion of tokens number).

TABLE IV: Execution time APE on MiBench (*automotive*) and PolyBench for CATREEN, ITHEMAL-tuned and MLPr

Model	CATREEN	ITHEMAL-tuned	MLPr
basicmath	**2.1%**	13.4%	67.1%
bitcount	**9.3%**	29.3%	68.3%
qsort	**19.0%**	37.5%	68.6%
susan:corner+edges	**18.9%**	35.3%	107.1%
susan:smoothing	**2.1%**	26.1%	67.1%
covariance	**15.8%**	18.4%	81.7%
gemm	**19.8%**	21.1%	30.2 %
gemver	**20.9%**	22.5%	79.9 %
gesummv	**15.0%**	15.3%	28.8 %
symm	**17.8%**	18.4%	35.0 %
syrk	**18.3%**	18.6%	30.0%
trmm	**22.6%**	24.8%	26.1%
2mm	**17.2%**	21.3%	25.3 %
3mm	**17.7%**	21.3%	27.2 %
atax	**23.2%**	24.8%	25.5%
bicg	**18.7%**	20.2%	28.1 %
dotigen	**21.5%**	25.5%	26.2 %
mvt	25.7%	**25.3%**	42.1%
cholesky	**20.6%**	22.4%	24.2 %
gramschmidt	**16.9%**	20.1%	21.8%
lu	**18.7%**	22.1%	106.5%
ludcmp	**11.0%**	16.9%	144.0%
trisolv	**9.1%**	11.9%	17.2%
floyd_warshall	**11.5%**	21.6%	268.7%
Avg.	**16.5%**	22.8%	62.2%

G. Inference time

The execution time of the predictions for CATREEN, ITHEMAL-tuned, and MLPr on a CPU is reported in Table V. We report the average time needed to infer the execution time of a BB in its sequence: we perform 1000 sequence inferences and average the results. As expected, MLPr is very fast as it does not need to process any kind of context. LSTM layers are much more complex than the MLP feed-forward layers.

CATREEN is slightly slower than ITHEMAL which is not surprising since it has an additional LSTM layer. Nevertheless, all timings are deemed low enough to be usable in practice.

TABLE V: inference timing

ML technique	MLPr	ITHEMAL	CATREEN
Average time(ms)	0.056	281.1	334.82

V. Conclusion

We have presented CATREEN, an ML-based program timing predictor. CATREEN leverages recurrent neural networks to predict the steady-state execution time of basic blocks in a program while taking into account the *execution context* formed by the previously executed basic blocks. Experimental results have shown that CATREEN's timing predictions are 18% and 27.6% better than those estimated by state-of-the-art context-agnostic ML-based tools on both synthetic and real programs respectively. [30] has observed that LSTM models can use a maximum of 200 context tokens which prevent them from learning longer-term information (i.e., they can remember sequences of hundreds but not thousands of items). An area for improvement, is to use *Transformers* [31] which have demonstrated a great ability to address text sequence learning. Using attention mechanisms, they can increase the quality of the contextual information drawn from the sequence under analysis and its size (yet, still limited to 1024 tokens at once when using BERT [32]). Nevertheless, transformers are not RNN and can only deal with fixed-length sequences. Therefore, the sequence has to be split into a certain number of segments or chunks before being used as input, causing *context fragmentation*. It may not be a problem for natural language processing where the text can be divided into paragraphs or sentences. However, in BBs execution time estimation, using Transformers naively would lose context execution or restrict its context length. [33], inspired by both the RNN and transformers mechanisms proposed the *Transformers XL*. They are a promising track for timing estimation but necessitate a huge amount of code data (much more than CATREEN) with high-performance GPUs to be trained from scratch for our problem.

References

[1] A. Tousi and M. Luján, "Comparative analysis of machine learning models for performance prediction of the spec benchmarks," *IEEE Access*, vol. 10, pp. 11 994–12 011, 2022.

[2] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, 2018.

[3] A. Ltd, "Cycle Models." [Online]. Available: https://www.arm.com/products/development-tools/simulation/cycle-models

[4] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Int. Conference on machine learning*. PMLR, 2019.

[5] D. Bruening and T. Garnett, "Building dynamic instrumentation tools with dynamorio," in *Proc. Int. Conf. IEEE/ACM Code Generation and Optimization*, 2013.

[6] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik, "Predicting execution time of computer programs using sparse polynomial regression," *Advances in neural information processing systems*, vol. 23, 2010.

[7] S. A. Seshia and J. Kotker, "Gametime: A toolkit for timing analysis of software," in *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011.

[8] F. Ritter and S. Hack, "Pmevo: portable inference of port mappings for out-of-order processors by evolutionary optimization," in *Proceedings of the 41st ACM SIGPLAN Conference*, 2020, pp. 608–622.

[9] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *26th Int. Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2017.

[10] M. Rapp, A. Pathania, T. Mitra, and J. Henkel, "Neural network-based performance prediction for task migration on s-nuca many-cores," *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1691–1704, 2020.

[11] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Int. Symposium on Code Generation and Optimization*. IEEE, 2007.

[12] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *14th ACM symposium on Principles and practice of parallel programming*, 2009.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] J. Wang, L.-C. Yu, K. R. Lai, and X. Zhang, "Investigating dynamic routing in tree-structured LSTM for sentiment analysis," in *Int. Conference on Empirical Methods in Natural Language Processing and Int. Joint Conference on NLP*, 2019.

[15] Y. Samih, S. Maharjan, M. Attia, L. Kallmeyer, and T. Solorio, "Multilingual code-switching identification via lstm recurrent neural networks," in *Proceedings of the Second Workshop on Computational Approaches to Code Switching*, 2016.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *4th IEEE international workshop on workload characterization*, 2001.

[18] L.-N. Pouchet, "The polyhedral benchmark suite," *On-line: http://www. cse. ohiostate. edu/pouchet/software/polybench*, 2012.

[19] Segger, "J-Trace PRO – The Leading Trace Solution." [Online]. Available: https://www.segger.com/products/debug-probes/j-trace/

[20] S. M. GmbH, "Ozone User Guide & Reference Manual," p. 348. [Online]. Available: https://www.segger.com/

[21] ARM, "ARM Compiler toolchain Assembler Reference." [Online]. Available: https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/instruction-summary

[22] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimaraes, and F. M. Q. Pereira, "Anghabench: A suite with one million compilable c benchmarks for code-size reduction," in *2021 IEEE/ACM Int. Symposium on Code Generation and Optimization*, 2021.

[23] PyTorch, "PyTorch." [Online]. Available: https://www.pytorch.org

[24] A. N. Amalou, I. Puaut, and G. Muller, "WE-HML: hybrid WCET estimation using machine learning for architectures with caches," in *27th IEEE Int. Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2021.

[25] D. MacNeil and C. Eliasmith, "Fine-tuning and the stability of recurrent neural networks," *PloS one*, vol. 6, no. 9, 2011.

[26] N. Jaques, S. Gu, R. E. Turner, and D. Eck, "Tuning recurrent neural networks with reinforcement learning," in *Int. conference on learning representations*, 2017.

[27] Z. Chen and Y. Yang, "Assessing forecast accuracy measures," *Preprint Series*, 2004.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd Int. Conference on Learning Representations (ICLR), San Diego, CA, USA*, 2015.

[29] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *2017 IEEE/ACM Int. Symposium on Code Generation and Optimization*. IEEE, 2017.

[30] U. Khandelwal, H. He, P. Qi, and D. Jurafsky, "Sharp nearby, fuzzy far away: How neural language models use context," *arXiv preprint arXiv:1805.04623*, 2018.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[33] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *arXiv preprint arXiv:1901.02860*, 2019.