Fast and Accurate Context-Aware Basic Block Timing Prediction using Transformers

Abderaouf Nassim Amalou
University of Rennes - Inria - CNRS - IRISA
Rennes, France
abderaouf.amalou@irisa.fr

Elisa Fromont
University of Rennes - IUF - Inria - CNRS - IRISA
Rennes, France
elisa.fromont@irisa.fr

Isabelle Puaut
University of Rennes - Inria - CNRS - IRISA
Rennes, France
isabelle.puaut@irisa.fr

Abstract

This paper introduces ORXESTRA, a context-aware execution time prediction model based on Transformers XL, specifically designed to accurately estimate performance in embedded system applications. Unlike traditional machine learning models that often overlook contextual information, resulting in biased predictions for individual isolated basic blocks, ORXESTRA overcomes this limitation by incorporating execution context awareness. By doing so, ORXESTRA effectively accounts for the processor micro-architecture without explicitly modeling micro-architectural elements such as caches, pipelines, and branch predictors. Our evaluations demonstrate ORXESTRA's ability to provide precise timing estimations for different ARM targets (Cortex M4, M7, A53, and A72), surpassing existing machine learning-based approaches in both prediction accuracy and prediction speed.

CCS Concepts: • **Computer systems organization** → *Embedded software.*

Keywords: Execution time estimation, Machine learning, long-short term memory, transformers model

ACM Reference Format:
Abderaouf Nassim Amalou, Elisa Fromont, and Isabelle Puaut. 2024. Fast and Accurate Context-Aware Basic Block Timing Prediction using Transformers. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3640537.3641572

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641572

1 Introduction

Over the past few years, numerous studies have explored the use of machine learning (ML) techniques for compiler optimization, specifically employing these techniques as cost functions that predict performance metrics such as execution time, energy consumption, and resource utilization [5, 21, 22, 27, 32]. These metrics are crucial for optimizing complex processors, where creating an accurate analytical model of the micro-architecture is often challenging, error-prone, or sometimes impossible due to the lack of detailed documentation and the necessary human expertise for model design.

The basic block[1] (BB) granularity is frequently used to profile program performance (execution time or energy consumption). However, existing techniques for BB timing estimation, for example, tools [22, 29], consider each BB in isolation, which results in biased solutions that do not account for the impact of pipelines, branch predictors, and cache memory, thereby lacking consideration of execution context in BB timing estimation. Some studies have attempted to address this limitation by incorporating context awareness into BB execution time estimation using stacked Long-Short Term Memory (LSTM) networks [5]. Nevertheless, LSTMs struggle with long sequences (exceeding 200 tokens, as empirically shown in [17]). More recently, transformer architectures have shown promise in overcoming the limitations of LSTMs, leading researchers to use them in binary analysis [20].

This paper presents ORXESTRA or cOntext-awaRe eXEcution Time eStimation using TRAnsformers. ORXESTRA accurately predicts the execution time of BBs within compiled binaries. Inspired by deep learning techniques commonly used in natural language processing, ORXESTRA uses a type of ML technique named *Transformers* [30], a particular attention-based architecture, to deliver fast and precise predictions. More specifically, our approach predicts execution time at the BB granularity using a *Transformers XL* [11], a recurrent variant of Transformers. The timing prediction of a BB uses the execution context of the BB (i.e., instructions executed before the BB under study). Furthermore, unlike LSTM-based models or traditional transformers, ORXESTRA

[1] A Basic Block (BB) is a straight-line sequence of instructions with no branches in except to the entry and no branches out except at the end.

provides accurate predictions for *long* basic blocks, typically present in real-world applications (e.g., see the study conducted in [27]) and long basic block sequences. ORXESTRA further includes automatic recognition of the structure of machine code (list of instructions, operands, and addressing modes), avoiding the cumbersome task of manually expressing this information from hardware documentation.

ORXESTRA has been tested on a range of Arm embedded processors, each with its own level of complexity. Although it can be used for the x86 architecture, we focused on testing our solution across architectures of varying complexity, and the ARM architecture was a good fit for this purpose. The tested processors include the Cortex-M4, a basic pipeline-only processor [3], the Cortex-M7 [4], which features a data cache, an instruction cache, and a branch predictor, the super-scalar Cortex-A53 [1], and the out-of-order Cortex-A72 [2]. Our experimental results demonstrate that ORXESTRA surpasses LSTM-based models [5, 22] and traditional Transformers-based solutions [20], showing a 28% improvement in estimation accuracy over the best competitor CATREEN (on average across all targets), while also being 98% faster in prediction timing than CATREEN [5].

ORXESTRA comes as a standalone tool that can be used directly by the application developer to profile programs' performance. In this case, its prediction efficiency allows ORXESTRA to replace costly performance measurement campaigns, either directly on the target processor or on a processor simulator. ORXESTRA can also be used as a companion tool for the compiler, allowing the compiler to quickly estimate the quality of the compilation optimizations adopted, for example, when using iterative compilation [10, 18].

The remainder of this paper is organized as follows: Section 2 presents an overview of related works that have utilized transformers or LSTMs to learn the representations of basic blocks (BBs) or their performance. Section 3 contains a comprehensive presentation of ORXESTRA, both regarding training and actual predictions. The dataset, competitors, processor targets, and experimental setup are described in Section 4. The performance of ORXESTRA during pretraining and fine-tuning is compared against state-of-the-art techniques in Section 5. Lastly, Section 6 concludes the paper.

2 Related Works

Performance estimation techniques using heuristics have been developed over the years to guide code optimization. Such techniques range from simple methods, such as counting the number of instructions, to complex machine learning (ML) techniques that predict the performance of code snippets, using, for example, multilayer perceptrons, recurrent neural networks such as Long Short-Term Memory (LSTM), and, more recently, transformers. This section provides an overview of works using ML-based techniques to estimate execution time for complex micro-architectures, as well as related research for automatic learning of the representation of machine code.

2.1 Execution Time Estimation ML-based Techniques

ITHEMAL [23] is the first model utilizing a hierarchical multi-scale RNN, specifically LSTM layers, to predict basic block performance, focusing on best-case execution time. It effectively captures interactions between instructions within the same basic block by sequentially processing each element (operations and operands). Alongside, the authors introduced BHive [9], a dataset of basic blocks for X86. However, BHive's data collection methodology, including isolating basic blocks, ensuring first-level cache memory accesses, and omitting branch instructions, does not truly represent processor execution. Our aim, in contrast to BHive, is to realize execution time representation authentic to the processor, enabling precise differentiation at the cycle level.

Granit [29] employs a Graph Neural Network (GNN) approach to model the dependencies between instructions within a basic block. This method relies on the user's expertise to define dependencies, which are then enforced by the GNN structure. In contrast, ORXESTRA adopts a matrix representation for depicting these dependencies. Specifically, it utilizes transformer-based mechanisms, where dependencies are learned through matrix attention techniques. ORXESTRA's approach is more autonomous, enabling it to uncover microarchitectural nuances without the need for predefined dependency structures by the user. This fundamental difference highlights ORXESTRA's ability to adaptively learn and discover intricate relationships within instruction sets, potentially offering a more efficient and user-independent pathway for modeling instruction dependencies.

DeepPM [27] in the same fashion as ITHEMAL [23] predicts the execution time of a basic block in isolation using a simplified Transformers architecture. The lack of details on the paper and the unavailability of the code made the reproduction of this solution impossible.

In contrast to these three previous works, CATREEN [5] relaxes the assumption that a basic block is executed in isolation and predicts the execution time of a basic block based on its actual context of execution, using three LSTM layers. However, ORXESTRA advances beyond this by employing Transformers XL, offering superior management of execution contexts, including support for longer sequences and more efficient handling of historical information. The differences between CATREEN and ORXESTRA, however, extend beyond just the choice of model. In the experiments Section 5, and for a fair comparison, we align CATREEN to the same configuration as ORXESTRA, but the two approaches differ in several key aspects:

- Input language: CATREEN uses a predefined input language where all memory addresses are constants,

while ORXESTRA learns its input language, treating each address as a separate token for better cache effect learning.
- Deployment: ORXESTRA is more practical to use than CATREEN as it automates the calculation of all execution contexts leading to a basic block, which is not the case in CATREEN.
- Dataset and targets: CATREEN has only been trained using a dataset formed of synthetic programs and on a single target (Cortex M7). In contrast, ORXESTRA uses a dataset formed of real programs and is trained on several different targets of varying complexity.

2.2 Learning Code Representation for Performance Estimation

Word2Vec [24] has been widely utilized in instruction representation learning, including performance models like ITHEMAL and CATREEN. However, there are notable limitations associated with Word2Vec: The absence of embeddings for out-of-vocabulary words poses a challenge when working with specialized languages, such as assembly code, where the vocabulary may contain domain-specific or rare words. In addition, Word2Vec treats each word as a single entity, disregarding its multiple meanings. This can lead to ambiguities in word representations, which becomes problematic when dealing with data dependencies in pipelines, vital for performance analysis. While Word2Vec has its merits and offers computational efficiency, Transformers have surpassed it in various aspects by employing more advanced techniques and models. Transformers provide enhanced contextualization, improved handling of long-range dependencies, and state-of-the-art performance. Palmtree [20] is a BERT-based [12] assembly encoder specifically designed to capture the characteristics of machine instruction sequences by generating general-purpose instruction embeddings through pre-training in assembly language. However, our work differs from Palmtree in two significant ways. Firstly, we employ a Transformers XL architecture to overcome limitations present in Transformers like BERT when dealing with sequence size. Secondly, we concentrate on the ARM assembly code instead of the Intel assembly code. Nevertheless, both Palmtree and ORXESTRA an be adapted to function with either Instruction Set Architecture (ISA). Other research efforts focusing on code representations, such as Graphcodebert [15] and CodeBert [13], are primarily interested in high-level code for tasks like code debugging, commenting, or code generation. In contrast, our research focuses on low-level code (which makes it a different language to understand, so it is not usable in our case) for performance estimation, aiming to improve performance estimation models for embedded systems.

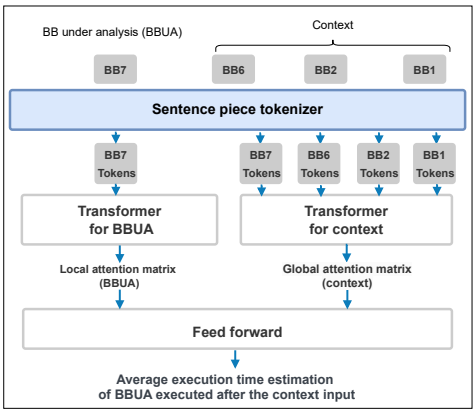

Figure 1. ORXESTRA Transformers XL-based architecture

3 Overview of ORXESTRA

ORXESTRA uses Transformer-XL (TXL) for performance prediction. Introduced in [30], transformers are neural architectures initially designed for Natural Language Processing (NLP) tasks like language translation and text summarization. They incorporate self-attention mechanisms, which enable the model to give appropriate importance to different parts of a sequence in the input data. This capability allows them to effectively capture the relationships between elements in a sequence and consider the overall context. However, the original Transformer architecture [31] has a fixed context window length, which limits its effectiveness for sequential data with long-term dependencies. This issue leads to 'context fragmentation,' where longer inputs are broken into smaller, independently processed fragments. Dai et al. [11] addressed this limitation by developing TXL, designed to more efficiently handle longer sequences.

3.1 Transformer XL

Transformers XL [11] marks a significant advancement in Transformers architectures by addressing the challenge of *context fragmentation*. One of its standout features is its capacity to "recall" or remember previously treated fragments of data. Instead of processing each fragment in isolation, Transformers XL integrates information from previous fragments, using this accumulated knowledge as a foundation when interpreting new data. This continuity is facilitated by a technique known as the "recurrence mechanism," which is somewhat akin to the workings of RNNs or LSTMs.

3.2 Architecture of ORXESTRA

ORXESTRA's architecture, as depicted in Figure 1, consists of several components. At its core, it utilizes a tokenizer called *Sentencepiece* [19]. A tokenizer is a common NLP function that transforms sequence elements into codes (often integers) that a neural network can process.

The entire system uses two TXLs, each serving a specific purpose. The first Transformer, displayed on the right side of the figure, focuses on creating an embedding representation of the execution context. The second transformer, depicted on the left side of the figure, takes care of processing the basic block under analysis (BBUA), as it holds the main information to predict the execution time.

When operating, the two transformers each produce an attention matrix, specifically a weight matrix-attention matrix of size $\#token \times \#token$. The size of $\#token$ is fixed at 512, which is the default setting for a small Transformer XL, as per Dai et al. [11]. When processing input, whether it is a context or a BBUA, the system divides it into smaller segments of 512 tokens. The Transformer XL's memory mechanism allows the system to retain information from the previously processed 512 tokens recursively, thus enabling the efficient processing of longer sequences. Padding is strategically applied at the beginning of each token sequence in the context input. This arrangement (e.g., BB1, BB2, BB6, and BB7 in the figure, with BB7 being the last one processed) ensures a "fresh" perspective on the BBUA. Consequently, the most recent information, especially about the BBUA, is prioritized in the final processing steps.

ORXESTRA generates two types of attention matrices: the *global attention matrix* for processing the context and the *local attention matrix* for the BBUA. These matrices are concatenated and then fed into a subsequent feed-forward neural network, as depicted at the bottom of the figure. This network is designed to estimate the average execution time (AET) associated with the BBUA.

3.3 Training of ORXESTRA

Training ORXESTRA consists of two main stages: pre-training each TXL (in practice, the same pre-trained model is used twice) and fine-tuning the entire architecture.

Pre-training: during the pre-training phase, the TXL is first trained using a self-supervised learning approach. Self-supervised pre-training is a technique often used in machine learning to leverage unlabeled data in order to "initialize" the weights of an (often large) neural network and help it learn better data representation to solve the final task. Training for this final task often involves a much smaller set of labeled data than the amount of data used during the pre-training phase. A pre-training supervised learning task is designed from the unlabeled data. For example, in Natural Language Processing (NLP), a standard pre-training task involves predicting a'masked' word in a sequence, where these masked words are randomly chosen. Importantly, the number of words to be masked is a hyperparameter, and in our approach, it is fixed to 15% of the sequence size. This learning task is thus supervised since the masked word is known from the model when parsing the sentence, but the label is generated automatically (randomly) without any human intervention (thus, it is called "self-supervised" learning). In our working context, our goal for this pre-training phase is to enable the model to understand the structure of assembly instructions presented in a textual format. This is also achieved by masking random operations or operands within the instruction sequence and training the model to predict them as output in a self-supervised training scheme. By doing so, we can leverage a very large (unlabeled) dataset consisting of thousands of disassembled binary programs to learn a trustworthy representation of assembly code (i.e., to have a relevant initialization of the weights of our TXL) that can be used for the final training phase. This final training phase consists of "fine-tuning" the pre-trained models.

Fine-Tuning: in the fine-tuning stage, ORXESTRA is trained to predict the execution time of individual BBs in context. In this phase, a specific set of programs, a target processor, and a measurement tool are employed. The execution times of basic blocks are measured (to obtain the training labels), and the corresponding instruction sequences are tokenized using the *Sentencepiece* technique presented before. To construct the training dataset, each BB's median execution time is considered instead of the mean (to filter the outliers), along with the tokenized BB itself and its associated context. The context size, which represents the number of basic blocks, serves as a hyperparameter for the TXL architecture.

3.4 Timing Predictions Using ORXESTRA

Figure 2 shows an overview of the deployment of ORXESTRA (e.g., its inputs and outputs). Once trained, using ORXESTRA requires performing three main steps:

1. CFG generation: the process of generating a control flow graph involves creating a CFG from the *binary* or *assembly code of programs*. Various binary analysis tools, such as Angr [28], can be used for this CFG generation.
2. Context extraction: to extract the sequence of BB forming the context, we currently ask the user to annotate each loop with its maximum number of iterations. footnoteBecause parametric loop handling is a complex problem itself. and use this information to unroll the loops. The degree to which we unroll these loops depends on the desired context size. As for function calls, they are virtually inlined during context extraction (recursivity is not handled in our case). Using the resulting CFG, we unroll loops and eliminate back edges, resulting in the creation of an acyclic graph. The edges in the graph are then reversed. Next, utilizing a depth-limited breadth-first search (BFS) algorithm, we locate paths from each basic block to its predecessors. The BFS algorithm takes the *context size* as input to limit the search. The resulting extracted context BB sequences are provided as inputs to ORXESTRA.

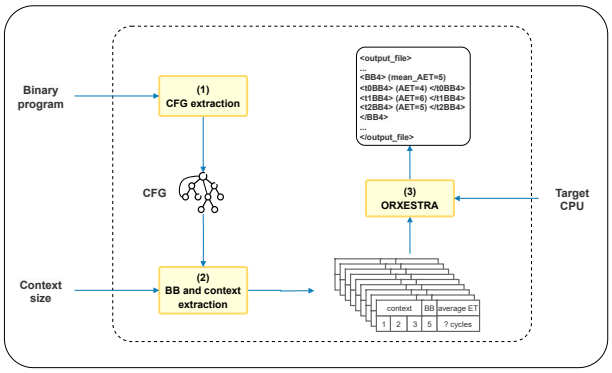

Figure 2. Timing predictions using ORXESTRA

3. Inference: In the final step, ORXESTRA loads the model based on the selected *target CPU* and proceeds to analyze all the given sequences of BB as inputs. It then predicts the execution time for the last BB of each sequence. If a BB appears at the end of multiple sequences, the BB is assigned the average execution time from all of these sequences. The final prediction gives the execution time among predictions for each BB.

4 Dataset, Competitors, Targets and Setup

We detail the data used to train (for both phases) and test ORXESTRA n Section 4.1. Section 4.2 describes the baseline methods used to assess the effectiveness of ORXESTRA The hardware and software setups are described in Section 4.3. Finally, Section 4.4 provides detailed information about the learning setup of ORXESTRA including all relevant hyperparameters.

4.1 Datasets and Benchmarks

As stated before, training ORXESTRA nvolves two primary steps: pre-training and fine-tuning. In the pre-training phase, which learns the structure of machine code, a large dataset of programs from CodeNet [25] is used. This dataset contains about 900,000 C programs obtained from public submissions on competitive programming websites. These programs undergo cross-compilation to the target architecture using *O3 optimization option* and are disassembled using the GNU binary tool *objdump*. From each assembly program, we extract relevant information such as instruction addresses and identification of BB limits (the start of a basic block and the end of it). It is important to note that the programs themselves are not executed during this process; they are used as a form of natural language data. The pre-training dataset is further used by *Sentencepiece* which performs its own learning phase and produces a model to tokenize our data. The *Sentencepiece* procedure progressively merges characters and character sequences, generating a vocabulary of smaller subword units based on statistical patterns. The model then uses these subword units to tokenize the input text, facilitating language modeling and addressing the issue of out-of-vocabulary. OOV in machine learning can have detrimental effects as it reduces generalization, results in information loss, triggers cascading errors, and presents difficulties in domain adaptation words. Once trained, *Sentencepiece* is able to tokenize binary programs written in the target instruction set, enabling efficient processing by the Transformers.

To fine-tune ORXESTRA a varied set of publicly available programs is employed[2], namely *The Algorithms* [3], *MiBench* [16], and *Polybench* [33]. Basic blocks and their respective contexts are extracted from these programs with a few modifications to obtain relevant data. For example, all instances of *printf* and system calls, which introduce redundant BBs and can potentially lead to overfitting, are eliminated from these programs. In Table 1, a summary of each benchmark suite is provided, including the number of programs in each dataset and the total count of basic blocks encountered during the execution of each program.

For more details, both the pre-training dataset and the fine-tuning dataset can be found in our public deposite [6].

Table 1. Composition of the dataset for the finetuning phase, showing benchmarks, the number of programs, and the total count of basic blocks retrieved per program. This dataset serves to the training and testing of all competitors also

Dataset name	Nb. of programs	Nb. of BB
The Algorithms	200	12123
PolyBench	30	11224
MiBench	14	8324
Total	244	31671

4.2 Baselines

ORXESTRA is compared to three context-agnostic timing predictors and one context-aware timing predictor. The first context-agnostic competitor is a Multi-Layer Perceptron (MLP) regressor, loosely referred to as a neural network (NN) [7]. Although not a naive approach, the neural network follows a feed-forward architecture that does not incorporate context information and further requires a fixed-size input. For our NN implementation, we input 233 static features from the basic blocks, mainly consisting of the proportions of various machine instruction types (e.g., MOV, ADD, LDR) with the associated access type (direct, indirect, or immediate). We use a greedy search algorithm to determine the optimal hyperparameters for the NN, including its number of hidden layers, the optimizer, the learning rate, and the loss function. Based on the validation dataset, the best parameters are: hidden layer sizes set to 512, 256, and 128; learning rate set to 'adaptive' with an initialization of 0.001; and use

[2]We also cross-compile this program using O3 option.
[3]Available here: https://github.com/TheAlgorithms/C

of the 'adam' solver. These hyper-parameters are coherent with what is used in [7].

Our second context-agnostic baseline is ITHEMAL [23], which uses LSTMs for execution time prediction. We re-implemented ITHEMAL from the original paper, porting the tokenization and embedding step of ITHEMAL to the ARM instruction set. Additionally, we tuned the model's hyperparameters to better fit the new data.

The third context-agnostic baseline is a re-implementation of BERT [12][4] here called "PalmTree ARM" [20]. This approach as Palmtree, involves pre-training BERT, and similarly to ORXESTRA using the masked language modeling pretext task specifically designed for ARM assembly code. Palmtree ARM takes a single basic block as input and predicts its timing hence without considering the execution context information. Our objective with this competitor is to compare ORXESTRA ith a Transformers model that performs similarly to ITHEMAL to assess the influence of the context awareness on the same type of neural architecture.

Finally, ORXESTRA s compared with its closest competitor, CATREEN, a context-aware execution time predictor introduced in [5]. We re-implemented CATREEN using the same hyperparameters as in the original study. Unlike ORXESTRA which relies on Transformers XL, CATREEN uses LSTMs which are recurrent neural architectures. Transformers XL excel at capturing dependencies using self-attention mechanisms, while LSTMs struggle due to the vanishing gradient problem when capturing long-range dependencies. Additionally, Transformers architectures are easier to design and train compared to LSTMs, even with pre-training.

In Table 2, a summary of the hyperparameters employed by all competitors is presented. To ensure a fair comparison, the context size parameter for both ORXESTRA nd CATREEN was selected to be the same.

Table 2. hyper-parameters used for ITHEMAL CATREEN, the PalmTree ARM, and ORXESTRA (NA: Non Applicable).

Hyperparameter	ITHEMAL	CATREEN	PalmTree ARM	ORXESTRA
Loss function	MAPE	sMAPE	MAPE	MAPE
Optimizer	'SGD'	'Adam'	'Adam'	'Adam'
Learning rate	0.001	0.0001	0.0001	0.0001
Embedding size	512	512	512	512
Feed forward structure and size	128	256, 256	512, 256, 128	512, 256, 128
Number of layers	2 LSTMs	3 LSTMs	6	4
Number of attention head	NA	NA	8	4
Memory length	NA	NA	NA	1024

[4]BERT (Bidirectional Encoder Representations from Transformers) is a type of language model based on the Transformers architecture.

4.3 Hardware and Software Setups

To obtain the timing values (labels) necessary to train ORXESTRA n the fine-tuning phase, we employ either a hardware-based or a software-based approach, depending on the availability of each solution. The hardware solution is always preferred when available for its negligible interference with execution (*probe effect*).

The hardware-based timing instrumentation leverages the Joint Test Action Group (JTAG) interface for the hardware solution and utilizes the J-Trace Pro trace solution from Segger [26]. This allows us to connect to the JTAG interface of the target processors, specifically the Cortex-M4 and Cortex-M7 in our case. To generate execution traces, we use Ozone [14], a cross-platform debugger and performance analyzer, in conjunction with J-Trace Pro. These traces provide valuable information such as the cycle counter value, instruction address, opcode, operands, and corresponding assembly code for each instruction.

The software-based solution involves adding instrumentation code to measure the execution time of individual BBs in a program. We retrieve the execution trace using GDB (the GNU Debugger) to obtain context and assembly code for the timed BB.

We measure the execution times of each basic block (BB) by running each program from each dataset using the default input. We intentionally start with a cold state of the processor and do not intervene during the program's execution to capture all possible microarchitecture aspects.

Our experiments encompass a variety of Arm processors, summarized in Table 3. The Cortex-M4 processor features a simple in-order pipeline with three stages and no cache. This processor enables us to validate our method on a deterministic processor with precise timing measurements obtained through the JTAG interface. The more advanced Cortex-M7 processor possesses a 6-stage in-order pipeline, data and instruction caches, and a branch predictor. The Cortex-A53 processor, hosted in a Raspberry Pi 3 features a dual issue 8-stage in-order pipeline, two levels of data and instruction caches, and a branch predictor. The Cortex-A72 processor, hosted in a Raspberry Pi 4 differs from the A53 through its out-of-order pipeline. Since the Cortex-A53 and Cortex-A72 lack a JTAG interface, we rely on reading the cycle counter register for timing measurements.

4.4 Setup for the Learning Phase

PyTorch was used to implement our model and the baseline ones. ORXESTRA as trained on a Tesla V100. Each setting (processor) required two days for ORXESTRA raining: 1,5 days for pre-training and 0,5 days to fine-tune the model. Perplexity[8][5] score was chosen as the value to optimize

[5]Perplexity is a measure of how well a probability model predicts a sample or a sequence of events. A lower perplexity indicates a better model fit to the data.

Table 3. Summary of the processors used and their micro-architectural features; I: Inordre processor, S: Superscalar, O: Out of Ordre processor.

Target	M4	M7	A53	A72
Measurement tool	JTAG	JTAG	Software	Software
OS?	Baremetal	Baremetal	Linux	Linux
Pipeline/#stages	I 3	I 6	I-S 8	O 15
Branch predictor	Yes	Yes	Yes	Yes
Cache memory	No	L1	L2	L2

during pre-training (see Equation 2) and Mean absolute percentage errors were used as loss during fine-tuning (see Equation 1). All the datasets (even in the pre-training phase) were split into training (70%), validation (10%), and test (the rest) sets containing different BBs. The MAPE (Mean Absolute Percentage Error) is also used to assess the performance of each model. It evaluates how far (as a percentage) the prediction is from the true timing. It is defined as:

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

$$\text{Perplexity}(D) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i)}} \quad (2)$$

Where:

- N is the total number of events or words in the dataset.
- w_i represents the ith event or word in the dataset.
- $P(w_i)$ is the probability assigned to event w_i by the probability model. It is the estimated likelihood of observing event w_i based on the model.

5 Experimental Results

Following the typical evaluation methodology of language models, ORXESTRA s evaluated in two ways: through *intrinsic evaluation* and *extrinsic evaluation*. Intrinsic evaluation involves assessing the model obtained at the end of the pre-training phase using specific unsupervised learning metrics (e.g. the perplexity score). It is provided in Section 5.1. On the other hand, extrinsic evaluation involves employing the models in actual tasks, such as the execution time estimation in our case, and evaluating their performance based on metrics such as the mean absolute percentage error (MAPE). This is provided in Section 5.2. The context-awareness feature of ORXESTRA s evaluated in Section 5.3, in which the impact of the context size on the estimations is investigated. Section 5.4 demonstrates that ORXESTRA xhibits superior scalability compared to other models when estimating the timing of large BBs. Section 5.5 study the robustness of the models to the unused optimization options. Lastly, Section 5.6, provides insights into the processing speed of each model by offering estimations of the number of instructions that can be processed per second.

5.1 Evaluation of the Pre-training

In the intrinsic evaluation experiment, both ORXESTRA nd the PalmTree ARM[6] were evaluated using the CodeNet [25] dataset. This evaluation assesses their performance in a masked language modeling task, specifically in recovering the masked operation/operand. The *perplexity* values for each model are provided in Table 4. The best results in the table are highlighted in bold. A lower perplexity score indicates that the language model is better at predicting masked words. Essentially, the perplexity score assesses how well each model predicts hidden words. Perplexity values, influenced by tokenization technique, dataset, and model, are represented as points. For intuitive interpretation, in English NLP research [11], a 5-point difference in perplexity is considered significant.

Table 4. Perplexity scores obtained by ORXESTRA nd the PalmTree ARM in the pre-training phase.

Target	M4	M7	A53	A72
PalmTree ARM	24.1	26.4	25.3	25.7
ORXESTRA	**19.2**	**23.1**	**22.2**	**21.8**

The results clearly indicate that ORXESTRA utperforms the PalmTree ARM across all targets. This observation is not surprising, as Transformers XL, unlike PalmTree ARM, exhibits superior memory capabilities for handling long sequences. In contrast, PalmTree ARM is limited by the restricted number of tokens that can be processed, which restricts their ability to capture dependencies beyond their specified context length. This hard sequence segmentation leads to context fragmentation, inefficient optimization, and ultimately a decline in performance.

5.2 Prediction Results on the Test Dataset

From our fine-tuning datasets, we extracted two distinct test sets for extrinsic evaluation purposes. The first test set consists of 500 BBs with fewer than 50 instructions and 500 BBs with more than 50 instructions. The second test set is created specifically for all the BBs whose timing can be successfully predicted by the PalmTree ARM. To accommodate the limitations of PalmTree ARM, which cannot handle BBs with token sequence sizes exceeding their capacity. This adjustment was essential to ensure a fair comparison between the models.

In Table 5, we use the first test dataset, and thus we do not provide the results for the PalmTree ARM (which could only perform predictions on the second test dataset and not all BB from the first set). The results include the Mean Absolute Percentage Error (MAPE), where lower percentages indicate

[6]A basic transformer called Palmtree [20] model trained for ARM ISA. We only compare to Palmtree as it already showed its superiority to other embedding models like word2vec [24].

better model performance. The best results in the table are highlighted in bold. Additionally, the Pearson correlation score is utilized as another evaluation metric to estimate how correlated the predictions are with the ground truth[7]. In this case, higher scores indicate better model performance.

Table 5. Test results of Neural Networks (NN), ITHEMAL [22], CATREEN [5], and ORXESTRA on various ARM Cortex targets: M4, M7, A53, and A72. The results are based on the first test dataset, which includes a balance between the number of small and large-sized BBs. Evaluation metrics: mean absolute percentage error (MAPE) and Pearson correlation (Corr.).

Scores	Target	M4	M7	A53	A72
MAPE	NN	26.4%	22.7%	38.4%	16.7%
	ITHEMAL	14.4%	17.6%	10.1%	11.4%
	CATREEN	8.8%	13.3%	8.5%	10.4%
	ORXESTRA	**7.8%**	**9.6%**	**5.2%**	**6.9%**
Corr.	NN	0.93	0.92	0.89	0.98
	ITHEMAL	0.90	0.90	0.98	0.98
	CATREEN	**0.99**	0.96	**0.99**	0.98
	ORXESTRA	**0.99**	**0.98**	**0.99**	**0.99**

Table 5 shows that ORXESTRA btains better MAPE performance than all other techniques for all the target architectures. The second best-performing model is CATREEN, which, like ORXESTRA considers the execution context of BBs. The worst-performing model is the Neural Network this shows the importance of accounting for the sequential information. The context-agnostic techniques, ITHEMAL and, as shown in Table 6 for the second test set, the PalmTree ARM are positioned after the context-aware techniques. The correlation is high for all models and better for the model designed to process sequential data.

The complexity of the target architecture plays a role in the final results, although its influence varies depending on the measurement method employed. When measuring timings on Cortex M4 and M7 using JTAG, we observe that errors on M7 are higher than for Cortex M4. This discrepancy is due to the deterministic nature of the Cortex M4 architecture, while M7 incorporates a cache with a random replacement policy, which introduces timing variability. However, this observation does not hold for more sophisticated architectures such as Cortex A53 and A72. For these processors, measurement methods involving software instrumentation were necessary, which introduced additional cycles into the measurements. The insertion of measurement instruments disrupts the execution, particularly affecting memory plans and cache behavior. As a result, the data (and in particular, the timing labels) obtained for these processors are slightly less accurate compared to processors with a JTAG interface. Consequently, making a direct comparison between these architectures is challenging.

[7]The ground truth is gathered using either the JTAG or instrumentation depending of the target processor.

Table 6 reports the results obtained on the second dataset. We can notice that the trends observed in the previous table 5 remain consistent.

Table 6. Test Results: Mean Absolute Percentage Error (MAPE) on Different Targets (M4, M7, A53, and A72) using the second test set. The Test Set is specifically chosen to be within the prediction capabilities of PalmTree ARM, ensuring a fairer comparison among models.

Target	M4	M7	A53	A72
Scores	MAPE	MAPE	MAPE	MAPE
Neural Networks	27.3%	24.5%	27.8%	25.9%
ITHEMAL	10.0%	14.4%	12.1%	13.0%
CATREEN	9.6%	14.5%	10.3%	11.8%
PalmTree ARM	9.1%	13.8%	13.5%	13.3%
ORXESTRA	**8.7%**	**6.8%**	**6.1%**	**7.5%**

5.3 Impact of the Context Size

In the previous Section, we saw the importance of context awareness to make accurate timing predictions (ORXESTRA nd CATREEN were the most accurate models thanks to this). However, an important question arises: how much context is necessary? To explore this, we conducted investigations on CATREEN and ORXESTRA or the different target architectures. Experimental results are reported in Table 7 and Table 8 respectively. The best results in the table are highlighted in bold.

Table 7. Impact of the context size (number of BB considered as context) on the Mean Absolute Percentage Error of CATREEN.

Target	M4	M7	A53	A72
None	13.0%	26.1%	36.3%	18.2%
1	12.5%	15.2%	21.0%	18.4%
3	**8.8**%	15.5%	**8.5**%	**10.4**%
6	9.3%	**13.3**%	12.5%	15.5%
20	10.2%	14.2%	9.5%	11.4%

Table 8. Impact of the context size (number of BB considered as context) on the Mean Absolute Percentage Error of ORXESTRA

Target	M4	M7	A53	A72
None	12.5%	24.5%	34.6%	13.3%
1	11.9%	14.6%	20.5%	13.1%
3	**7.8**%	14.5%	22.9%	14.5%
6	8.8%	**9.6**%	**5.2**%	**6.9**%
20	9.2%	13.7%	8.3%	8.8%

The results highlight the importance of context in timing estimates. Both CATREEN and ORXESTRA when deprived of context, produce estimates akin to standard neural networks. As the context size grows, the prediction errors diminish. Yet, there's a limit to this improvement. For ORXESTRA errors stabilize after including 3 BBs for the M4 architecture and 6 BBs for other processors. For CATREEN, most processors stabilize at 3 BBs, but M7 needs 6 BBs. This plateau in error reduction can be attributed to the inherent constraints of LSTM architectures. Overly long contexts can overload the context vector, making it less effective within the set hyperparameters of both models. In future research, we intend to delve deeper into this phenomenon, necessitating significant computational resources to gain further insights and confirm our intuition.

5.4 Impact of the Basic Block Size

Figure 3 shows the mean absolute error[8] for six sets of 500 basic blocks (BBs) in the test dataset, categorized by instruction count: under 10, 10–20, 20–30, 30–40, 40–50, 50–100, and over 100 instructions. ORXESTRA enerally shows lower errors across BBs of varying sizes and architectures. CATREEN and ITHEMAL's error increases significantly for BBs over 100 instructions. This confirms that LSTM-based models scale poorly, even in a task like predicting the execution time of basic blocks, probably due to the vanishing gradient problem. We also observe that the ITHEMAL errors are higher than CATREEN's error, which is surely due to the context management, which will be more influential for larger basic blocks that have a higher probability of making more memory accesses and being more dependent on the context than small-sized basic blocks. As deducted in Section 5.2, ORXESTRA as lower average errors for complex architectures like Cortex A53 and A72, but higher errors for in-order pipeline architectures. Analysis of execution time standard deviations reveals lower variability in more advanced architectures (M4: 252 cycles, M7: 113 cycles, A53: 83 cycles, A72: 64 cycles), which likely contributes to reduced prediction errors.

5.5 Optimization Effect on Prediction

In this experiment, we want to use different GCC optimization levels (O0, O1, O2, O3) to compile and generate the test dataset. This allows for an investigation into how sensitive the models are to changes in optimization levels. The goal is to answer the question: are the models robust enough to maintain performance across different optimization levels? Table 9 gives the results of this experimentation, and we observe that ORXESTRA onsistently outperforms other models across all targets and optimization levels, demonstrating its robustness. In contrast, Neural Networks exhibit the highest error in results, and the highest in term of variability making them the most sensitive to compiler optimization. ITHEMAL consistently outperforms Neural Networks and occasionally surpasses CATREEN.

[8]Mean Absolute Error (MAE) is calculated as $\frac{1}{n} * \sum_{1}^{i=n} |prediction_i - truth_i|$.

5.6 Inference Throughput

Table 10 displays the instruction rate per second achieved by each machine learning model. Interestingly, this time is independent of the complexity of the target processor architecture, so we report the average over all processors. The throughput calculation is based on the 1000 basic blocks utilized in the previous experiments (the first test set). To ensure a fair comparison, we present the results in the first column with a batch size of 1, followed by the results with a batch size of 32 in the second column for all techniques. Notably, neural networks demonstrate the highest execution speed, despite their lower accuracy. PalmTree ARM, which does not consider the execution context, follows closely. ORXESTRA which processes this context. Both Transformers based solutions provides a better execution speed compared to LSTM-based networks (ITHEMAL and CATREEN), which require sequential processing of each instruction. Consequently, CATREEN is the slowest among them due to the additional context processing involved.

Table 10. The mean throughput over all processors, when treating 1000 BB for each technique (with a batch size of 1 and batch size of 32).

	Throughput Instruction/second	
Batch size	1	32
Neural Networks	5131	162140
ITHEMAL	1627	45379
CATREEN	1356	32644
PalmTree ARM	3809	112468
ORXESTRA	2691	74172

6 Conclusion

ORXESTRA s a tool that estimates the timing of basic blocks within a program. It takes into account the execution context formed by previously executed basic blocks. Experimental results have shown that its timing predictions are 28% better than those of state-of-the-art context-aware LSTM-based CATREEN, while being 98% faster than the latter. Moving forward, we plan to develop a larger model that can incorporate all execution contexts, to capture memory accesses in the control flow graph and represent them as an embedding. Introduce explainability for a more accurate results analysis and create a faster model using transfer learning.

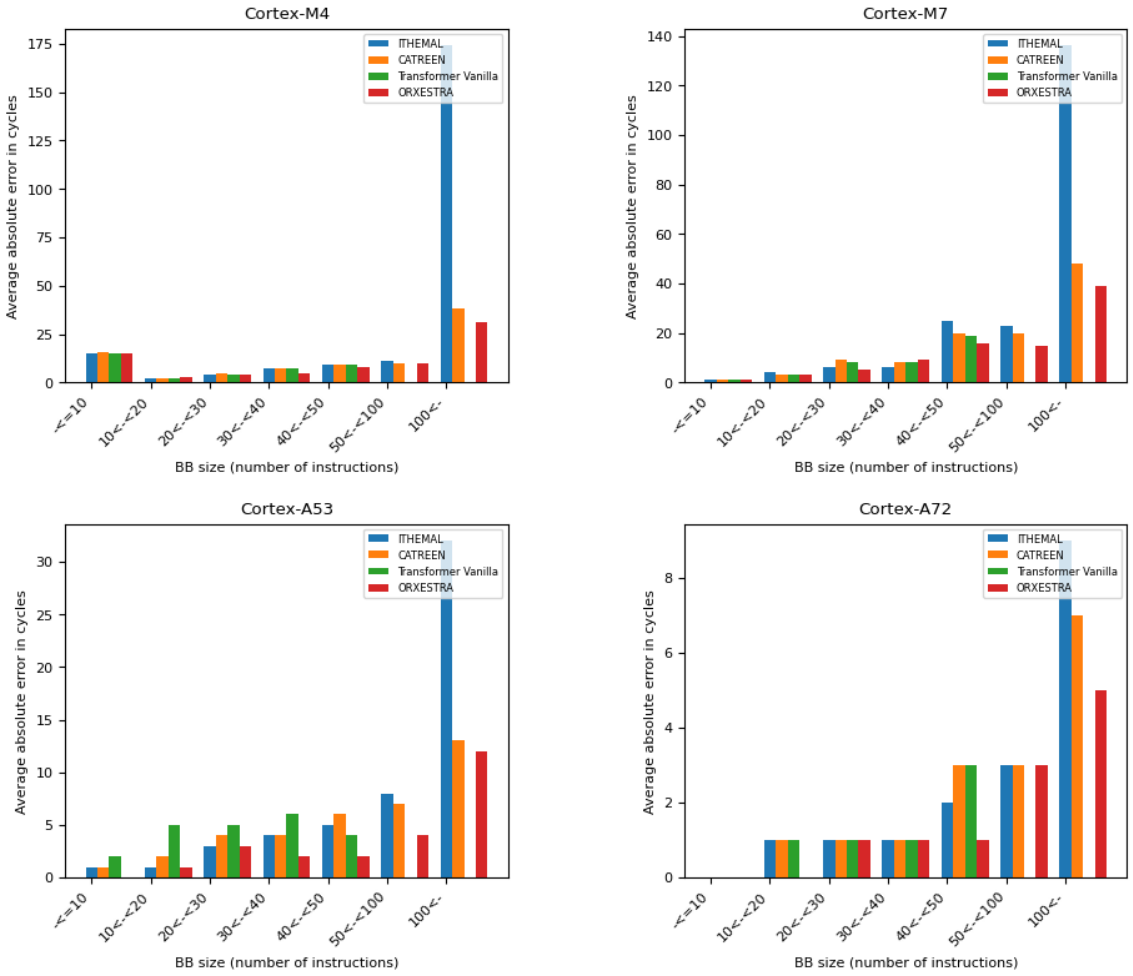

Figure 3. Mean absolute average cycle error boxplot comparison of ITHEMAL (blue), CATREEN (orange), PalmTree ARM (green), and ORXESTRA red) for different processors (M4, M7, A53, A72) and six Categories of basic blocks. The most left category represents basic blocks with a size of 10 or less instructions (-<=10), while the most right category includes basic blocks with a number of instructions exceeding 100 instructions (100<). Each subfigure represents a processor.

Table 9. MAPE performance of ORXESTRA, CATREEN, ITHEMAL and Neural Networks across various GCC optimization levels (O0, O1, O2 and O3) and architectural targets

Target	M4				M7				A53				A72			
Optimization	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3
Neural Networks	28.1%	23.2%	21.9%	26.4%	21.1%	19.9%	28.3%	22.7%	25.2%	40.4%	37.1%	38.4%	18.2%	17.1%	16.4%	16.7%
ITHEMAL	14.1%	14.5%	14.5%	14.4%	18.2%	18.2%	17.7%	17.6%	9.1%	11.0%	10.6%	10.1%	12.2%	12.3%	11.5%	11.4%
CATREEN	8.9%	8.2%	8.9%	8.8%	13.4%	13.1%	12.8%	13.3%	9.7%	9.7%	9.2%	8.5%	11.1%	11.4%	10.8%	10.4%
ORXESTRA	**7.6%**	**7.7%**	**7.7%**	**7.8%**	**8.9%**	**8.2%**	**9.9%**	**9.6%**	**6.2%**	**6.1%**	**6.3%**	**5.2%**	**7.9%**	**7.2%**	**7.4%**	**6.9%**

References

[1] Accessed 2023. ARM Cortex-A53 Processor. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53.

[2] Accessed 2023. ARM Cortex-A72 Processor. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a72.

[3] Accessed 2023. ARM Cortex-M4 Processor. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4.

[4] Accessed 2023. ARM Cortex-M7 Processor. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m7.

[5] Abderaouf N Amalou, Elisa Fromont, and Isabelle Puaut. 2022. CATREEN: Context-Aware Code Timing Estimation with Stacked Recurrent Networks. In *2022 IEEE 34th International Conference on Tools with Artificial Intelligence (ICTAI)*. 571–576. https://doi.org/10.1109/ICTAI56018.2022.00090

[6] Abderaouf Nassim AMALOU, Isabelle Puaut, and Elisa Fromont. 2023. *Pre-training and fine-tuning dataset for transformers consisting of basic blocks and their execution times (average, minimum, and maximum) along with the execution context of these blocks, for various Cortex processors M7, M4, A53, and A72.* https://doi.org/10.5281/zenodo.10043908

[7] Abderaouf N Amalou, Isabelle Puaut, and Gilles Muller. 2021. WEHML: hybrid WCET estimation using machine learning for architectures with caches. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 31–40. https://doi.org/10.1109/RTCSA52859.2021.00011

[8] Stanley F Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language* 13, 4 (1999), 359–394.

[9] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sýkora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 167–177. https://doi.org/10.1109/IISWC47752.2019.9042166

[10] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. 2012. Deconstructing iterative optimization. *ACM Trans. Archit. Code Optim.* 9, 3, Article 21 (oct 2012), 30 pages. https://doi.org/10.1145/2355585.2355594

[11] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[14] SEGGER Microcontroller GmbH. [n. d.]. Ozone User Guide & Reference Manual. , 348 pages. https://www.segger.com/

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode BERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[16] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14. https://doi.org/10.1109/WWC.2001.990739

[17] Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. 2018. Sharp nearby, fuzzy far away: How neural language models use context. *arXiv preprint arXiv:1805.04623* (2018).

[18] Peter M. W. Knijnenburg, Toru Kisuki, and Michael F. P. O'Boyle. 2002. Iterative Compilation. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS (Lecture Notes in Computer Science, Vol. 2268)*, Ed F. Deprettere, Jürgen Teich, and Stamatis Vassiliadis (Eds.). Springer, 171–187. https://doi.org/10.1007/3-540-45874-3_10

[19] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[20] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 3236–3251. https://doi.org/10.1145/3460120.3484587

[21] Martin Maas. 2020. A Taxonomy of ML for Systems Problems. *IEEE Micro* 40, 5 (2020), 8–16. https://doi.org/10.1109/MM.2020.3012883

[22] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*. PMLR, 4505–4515.

[23] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *Int. Conference on machine learning*. PMLR.

[24] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[25] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).

[26] Segger. [n. d.]. J-Trace PRO – The Leading Trace Solution. https://www.segger.com/products/debug-probes/j-trace/

[27] Jun S. Shim, Bogyeong Han, Yeseong Kim, and Jihong Kim. 2022. DeepPM: Transformer-based Power and Performance Prediction for Energy-Aware Software. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1491–1496. https://doi.org/10.23919/DATE54114.2022.9774589

[28] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

[29] Ondřej Sýkora, Phitchaya Mangpo Phothilimthana, Charith Mendis, and Amir Yazdanbakhsh. 2022. GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 14–26. https://doi.org/10.1109/IISWC55918.2022.00012

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[32] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[33] Tomofumi Yuki and Louis-Noël Pouchet. 2016. PolyBench 4.2. 1 (pre-release).

Received 13-NOV-2023; accepted 2023-12-23