RESEARCH-ARTICLE

DeepPM: Predicting Performance and Energy Consumption of Program Binaries Using Transformers

JUN S SHIM, Seoul National University, Seoul, South Korea

HYEONJI CHANG, Seoul National University, Seoul, South Korea

YESEONG KIM, Daegu Gyeongbuk Institute of Science and Technology, Daegu, South Korea

JIHONG KIM, Seoul National University, Seoul, South Korea

Open Access Support provided by:

Seoul National University

Daegu Gyeongbuk Institute of Science and Technology

Published: 17 October 2025
Online AM: 28 March 2025
Accepted: 10 March 2025
Revised: 07 March 2025
Received: 01 August 2024

DeepPM: Predicting Performance and Energy Consumption of Program Binaries Using Transformers

JUN S. SHIM, Dept. of Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of)
HYEONJI CHANG, Dept. of Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of)
YESEONG KIM, Electrical Engineering and Computer Science, DGIST, Daegu, Korea (the Republic of)
JIHONG KIM, Dept. of Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of)

Accurate estimation of performance and energy consumption is critical for optimizing application efficiency on diverse hardware platforms. Traditional methods often rely on profiling and measurements, requiring at least one execution, making them time-consuming and resource-intensive. This article introduces the Deep Power Meter (DeepPM) framework, leveraging deep learning, specifically the Transformer architecture, to predict performance and energy consumption of basic blocks directly from compiled binaries, eliminating the need for explicit measurement processes. The DeepPM model effectively learns the performance and energy consumption of basic blocks, enabling accurate predictions for each. Furthermore, the framework enhances applicability across different ISAs and microarchitectures, addressing limitations of state-of-the-art ML-based techniques restricted to specific processor architectures. Experimental results using the SPEC CPU 2017 benchmark suite show that DeepPM achieves significantly lower prediction errors compared to state-of-the-art ML-based techniques, with a 24% improvement in performance and an 18% improvement in energy consumption for x86 basic blocks, and similar gains for ARM processors. Fine-tuning with minimal data from the Phoronix Test Suite further validates DeepPM's robustness, achieving an error of approximately 13.7%, close to the fully trained model's 13.3% error. These findings demonstrate DeepPM's ability to enhance the accuracy and efficiency of performance and energy consumption predictions, making it a valuable tool for optimizing computing systems across diverse hardware environments.

CCS Concepts: • **Hardware → Timing analysis**; **Power estimation and optimization**; **Software tools for EDA;**

Additional Key Words and Phrases: Performance estimation, energy consumption estimation, basic block, deep learning, transformer

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) through grants from the Korea government (MSIT) (RS-2024-00459026 and RS-2024-00456287), and by Samsung Electronics Co., Ltd (IO221213-04119-01). Yeseong Kim was supported by IITP through a grant from the Korean government (MSIT) (No.2022-0-00991). The ICT at Seoul National University provided research facilities for this study.
Authors' Contact Information: Jun S. Shim, Dept. of Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of); e-mail: jsshim@davinci.snu.ac.kr; Hyeonji Chang, Dept. of Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of); e-mail: hyeonji@davinci.snu.ac.kr; Yeseong Kim (Corresponding author), Electrical Engineering and Computer Science, DGIST, Daegu, Korea; e-mail: yeseongkim@dgist.ac.kr; Jihong Kim (Corresponding author), Dept. of Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of); e-mail: jihong@davinci.snu.ac.kr.

ACM 1084-4309/2025/10-ART107
https://doi.org/10.1145/3725887

ACM Reference Format:
Jun S. Shim, Hyeonji Chang, Yeseong Kim, and Jihong Kim. 2025. DeepPM: Predicting Performance and Energy Consumption of Program Binaries Using Transformers. *ACM Trans. Des. Autom. Electron. Syst.* 30, 6, Article 107 (October 2025), 27 pages. https://doi.org/10.1145/3725887

1 Introduction

Understanding the performance and energy behaviors of applications running on target hardware platforms is a fundamental requirement for enhancing system efficiency. Accurate estimations of performance and energy consumption not only guide the selection of more efficient code variants at design and compile time [16, 28], but also facilitate optimal runtime management through techniques such as voltage and frequency scaling [29, 34]. These estimation techniques provide crucial insights into system design, aiding in achieving a balance between performance and energy efficiency within limited resources.

Researchers have extensively explored techniques for predicting performance and energy consumption across various levels of program and system design [9, 10, 19, 23, 26, 27, 31]. One prominent approach involves analyzing source code or compiled binaries, partitioning the target program workload into smaller slices, which represent *high-level granular components* such as basic blocks, functions, or phases. This approach allows for constructing analytical models using the profiled information for each segmented element, thus facilitating the analysis of software execution flows. Compared to lower-level granularity techniques like gate/circuit/RTL (Register-Transfer Level)/instruction level simulations or analytical models, this high-level abstraction significantly reduces computational complexity and resource usage in estimating performance and energy consumption.

However, these approaches come with significant overheads due to the necessity of detailed per-component profiles. Many analytical models rely on **Performance Monitoring Counter** (**PMC**) events as input features to predict performance or energy consumption accurately [7, 18, 30]. Consequently, users must execute the segmented components or target program *at least once* on a fine-grained simulator or real hardware to gather PMC measures and relevant events. This process is both time-consuming and restrictive, as it requires feasible measurement or profiling scenarios. The reliance on detailed measurements presents a major challenge for employing techniques based on high-level abstraction, necessitating a more efficient solution.

In this article, we propose the **Deep Power Meter** (**DeepPM**) framework, which predicts performance and energy consumption for basic blocks efficiently and accurately, using only compiled binaries and without any measurement process. To enable prediction using only compiled binaries, we employ deep learning techniques, specifically, the Transformer architecture [32], originally designed for **Natural Language Processing** (**NLP**). In our approach, DeepPM treats instruction sequences as textual data in NLP, predicting core cycle and energy values consumed during the execution of the sequence without actual execution.

Our work focuses specifically on prediction at the basic block level, a widely adopted approach for high-level abstraction [2, 17, 29, 33, 35]. A basic block is formally defined as a contiguous sequence of instructions within a program, typically demarcated by branch instructions during the compilation process. This definition ensures that each basic block encapsulates a distinct function, and their connections effectively represent the entire program, making them ideal for high-level abstraction techniques.

The DeepPM framework extracts basic blocks from the user-input target program and tokenizes them with the sophisticatedly designed DeepPM tokenizer, feeding them into the DeepPM model for performance and energy consumption prediction. Since the trained model only gets the

basic block itself as input, it completely eliminates the need for explicit measurement processes. Consequently, users can obtain accurate estimations of performance and energy consumption for previously unseen basic blocks without the necessity of running them on fine-grained simulators or real hardware.

We demonstrate that the DeepPM model architecture effectively learns the performance and energy consumption of these basic blocks, enabling the trained model to make accurate predictions for each. Furthermore, we enhance the applicability of the DeepPM framework across different **Instruction Set Architectures** (**ISAs**) and microarchitectures, addressing the limitations of prior works that are restricted to specific processor architectures [12, 22]. Existing studies develop models targeting specific ISAs, necessitating full retraining for each processor. This process demands significant effort in data collection and training time. In contrast, the tokenizer devised for the DeepPM framework identifies commonalities across various ISAs, allowing the model to generalize effectively across different processor architectures, such as x86 and ARM. Our fine-tuning approach further refines the model to account for variations in microarchitectures within the same ISA, reducing the need for extensive retraining. This cross-architecture generalization capability significantly enhances the practical utility of the DeepPM framework, making it robust for performance and energy consumption prediction across diverse computing environments.

To evaluate the effectiveness of the DeepPM framework, we conducted extensive experiments using the SPEC CPU 2017 benchmark suite [6], labeled with performance and energy consumption measurements from both x86 and ARM processors. We compare the DeepPM models with those trained using the Ithemal architecture [22], a state-of-the-art ML-based technique for predicting the performance of x86 basic blocks employing an LSTM-based architecture.

Our evaluation results demonstrate that the DeepPM framework achieves superior accuracy in predicting performance and energy consumption for both x86 and ARM basic blocks. Specifically, for the x86 cases, the DeepPM model exhibited approximately 24% lower error in performance and 18% lower error in energy consumption compared to the Ithemal model. Similarly, for the ARM processors, the DeepPM model showed significant reductions in prediction errors, underscoring its robustness across different ISAs. Notably, the DeepPM model excelled in predicting long basic blocks (e.g., more than 50 instructions), which constitute a majority of actual program executions. In the longest basic block group, the DeepPM model achieved an 88% improvement in performance prediction and a 73% improvement in energy consumption prediction for the x86 cases, with substantial improvements observed for the ARM processors as well.

We also evaluated the fine-tuning capability of the DeepPM framework across different microarchitectures within the same ISA. For these experiments, we utilized applications from the Phoronix Test Suite [21] to extract a small set of basic blocks labeled with performance data. We fine-tuned the DeepPM model, originally trained on a full set of SPEC CPU 2017 basic blocks from one microarchitecture, using this small Phoronix dataset collected on another microarchitecture. The fine-tuned model achieved an error of approximately 13.7%, closely matching the 13.3% error of the fully trained model. This result highlights that our proposed models can serve as *pre-trained foundation models*, allowing users to fine-tune them with minimal data for their specific microarchitectures while maintaining high prediction accuracy.

This article makes the following key contributions:

— We introduce the DeepPM framework, which enables efficient and accurate prediction of performance and energy consumption solely from compiled binary code, eliminating the need for detailed measurement processes.
— We present a deep learning-based prediction approach that translates instruction sequences of a basic block into performance and energy consumption estimates, utilizing a novel

Transformer architecture with a tailored tokenizer design that enhances adaptability across various ISA and microarchitectures.
— We demonstrate the effectiveness of the DeepPM framework in predicting long basic blocks, which constitute the majority of actual program executions, thereby significantly improving prediction accuracy for complex sequences.
— We show the cross-architecture generalization capability of the DeepPM framework through extensive experiments, highlighting its robustness across x86 and ARM processors and demonstrating the applicability of fine-tuning for different microarchitectures.

The rest of this article is organized as follows: Section 2 reviews previous works on predicting the performance and energy consumption of basic blocks using learning techniques, introduces the key motivation behind our work, and provides background on the Transformer architecture and transfer fine-tuning. Section 3 introduces the DeepPM framework and provides an overview of its entire workflow. In Section 4, we present the DeepPM tokenizer, which effectively tokenizes basic blocks across various ISAs without losing generality. Section 5 details our DeepPM model architecture, which learns the system behaviors of basic blocks within the Transformer model architecture. Section 6 elaborates our experimental setup, including the detailed methods of basic block extraction and labeling of performance and energy data. Finally, in Section 7, we evaluate the proposed DeepPM framework, discussing its prediction quality and usability across various ISAs.

2 Background and Motivation

2.1 Learning-based Basic Block Performance/Energy Consumption Estimation

Recent research has proposed machine learning-based techniques to predict the cycle and energy consumption of basic blocks without needing their physical execution [12, 22]. These techniques use assembly-written basic blocks as input to the model, which is trained to predict the cycle or energy consumption through supervised learning.

One such work, presented in [12], estimates the energy consumption of basic blocks using an **Artificial Neural Network** (**ANN**) architecture with instruction type-based tokenization. In this approach, basic blocks are segmented or padded with null values to conform to the fixed input size of ANNs. Each instruction is converted into a token based on its type. The tokenized data are then used as input for the neural network models.

Another notable study, Ithemal [22], employs a **Long Short-Term Memory** (**LSTM**)-based model architecture to predict the cycle count of basic blocks. It utilizes an Ithemal tokenizer to input basic blocks into the model. The LSTM model is specialized for sequence data, capable of handling varying input lengths and learning order information. The Ithemal tokenizer represents each instruction as a list of tokens, distinguishing various components such as the opcode, source/destination operands, register, base memory, and constants. This tokenization method allows the Ithemal model to distinguish variations within an instruction as well as between different instructions, ensuring a detailed and accurate representation of each basic block for the learning procedures.

While these studies demonstrate the feasibility of machine learning for performance and energy consumption estimation, they pose several challenges in real-world applications. For instance, the ANN-based approach in [12] faces limitations due to its fixed input size, which fails to accommodate the variability in the number and order of instructions within basic blocks. This approach also does not capture the sequential nature of instructions, and the instruction type-based tokenization overlooks operand variations and instruction order, missing critical sequence information. Additionally, this method necessitates creating new tokenizers for different ISAs, complicating its application across various processors.

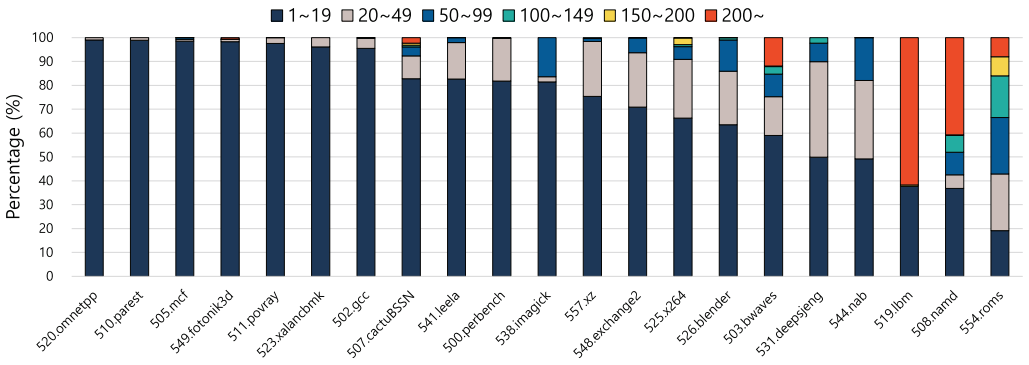


Fig. 1. Runtime usage ratio per group during the execution of the SPEC CPU 2017 benchmark suite.

On the other hand, the Ithemal model [22] exhibits reduced accuracy when dealing with very long sequences. This limitation leads to potential information loss and challenges in learning long-term dependencies. Our experiments indicate that long basic blocks, which are frequently utilized in actual program execution, are not effectively handled by the LSTM-based Ithemal model. Moreover, Ithemal primarily targets x86 basic blocks, with a tokenizer tailored to distinguish internal elements of instructions specific to the x86 architecture, which limits its general applicability across different processors.

2.2 Basic Block Length vs. Prediction Quality

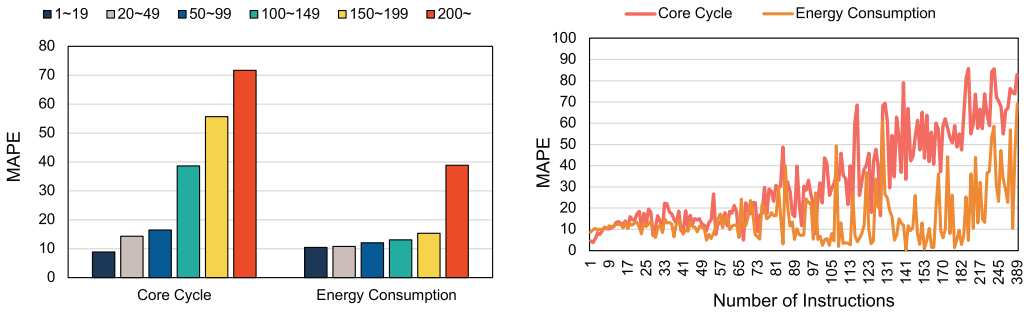
To examine the lengths of basic blocks executed in real-world applications, we conducted an experiment using 21 workloads from the SPEC CPU 2017 benchmark [6]. Using the Pin tool [20], we traced all the basic blocks executed during the runtime of each workload. We counted the number of instructions in each basic block and determined the frequency of each instruction count.

We categorized the data into six groups to determine the usage ratio for each group, as presented in Figure 1. The results show that the shortest basic block group (1–19 instructions) constitutes the largest portion of the total executions in most applications, except for three (519.lbm, 508.named, and 554.roms). In seven applications (520.omnetpp, 510.parest, 505.mcf, 549.fotonik3d, 511.povray, 523.xalancbmk, and 502.gcc), this group accounts for more than 90% of the executions. This observation shows the importance of accurately predicting short basic blocks, as they form a significant proportion of the blocks encountered during the execution of most applications.

Despite their lower frequency, longer basic block groups significantly impact overall program execution, making their accurate prediction equally important. For instance, in nine applications (538.imagick, 525.x264, 526.blender, 503.bwaves, 531.deepsjeng, 544.nab, 519.lbm, 508.namd, and 554.roms), more than 10% of the basic blocks contain over 50 instructions. Notably, in three of these applications (519.lbm, 508.named, and 554.roms), over 50% of the basic blocks executed have more than 50 instructions.

These findings highlight the importance of accurately predicting both the more frequent shorter basic blocks and the longer ones. To evaluate how effectively existing methods estimate the performance and energy consumption of long basic blocks, we trained the LSTM model based on the state-of-the-art Ithemal approach [22] using a training dataset and assessed its estimation quality on a test dataset not used during training. Detailed descriptions of our datasets and training process are provided in Sections 6 and 7.

Figure 2 presents the **Mean Absolute Percentage Error** (**MAPE**) values for each group of basic blocks. The results indicate that the model achieves high accuracy for the shortest basic



(a) Average MAPE per Group (b) Average MAPE per Instruction Length

Fig. 2. The test results of the LSTM-based model trained on the basic block datasets obtained from the SPEC CPU 2017 benchmark suite.

block group (1–19 instructions). However, as the length of the basic block groups increases, the model's prediction quality severely degrades. For the longest basic block groups, the method exhibited significant errors, e.g., up to 71.63% and 38.82% for performance and energy consumption, respectively.

As shown in Figure 1, groups other than the shortest ones also constitute a significant proportion in practical applications. Consequently, the trend of increasing error with the number of instructions further exacerbates the prediction quality for the overall workload analysis, as shown by the average MAPE for each instruction count in Figure 2(b). These experimental results confirm that while long basic blocks are frequently utilized in real applications, the prediction quality of the LSTM-based Ithemal model decreases as the length of the basic blocks increases.

Therefore, we conclude that existing techniques often struggle to provide accurate performance and energy estimates for longer basic blocks, despite their crucial role in certain applications where they account for a substantial portion of the overall execution. Addressing this limitation is one of our primary goals in designing the DeepPM framework, which aims to accurately predict the performance and energy consumption of long basic blocks.

2.3 Transformer Architecture

The Transformer architecture [32] has revolutionized the field of NLP, leading to the development of various model structures that demonstrate exceptional performance across numerous language tasks. At the core of the Transformer is the attention mechanism, which calculates relationships between tokens in the input sequence. This mechanism allows the model to consider the context of each token in relation to all other tokens, providing a powerful way to capture dependencies across the entire sequence.

Unlike traditional models like LSTM networks that process input sequentially and typically compute relationships in a single direction, the Transformer processes the entire input simultaneously. This parallel processing not only reduces training time but also enables the model to handle long input sequences effectively by processing all tokens at once, making it particularly advantageous for tasks involving lengthy sequences.

The Transformer consists of an encoder and a decoder, each with distinct roles. The encoder processes the input sequence and creates a vector representation, while the decoder takes this representation and generates the output sequence. In our work, we focus primarily on the encoder, as our goal is to utilize vector representations rather than generating new sequences. The Transformer encoder is composed of multiple layers, each containing a self-attention mechanism and a

feed-forward neural network, along with layer normalization and residual connections. Each layer receives the entire input sequence, computes the self-attention and feed-forward transformations, and passes the resulting vector representations to the next layer. This iterative process refines the representation at each step.

These meaningful vector representations enable the Transformer to handle various NLP tasks such as machine translation, text summarization, sentiment analysis, and question answering. The versatility of the Transformer has also extended its applications beyond NLP to fields like **computer vision** (**CV**), where it has been successfully used for tasks such as image classification, object detection, and image generation. In this article, leveraging the Transformer's ability to effectively process long input sequences and capture complex dependencies, the DeepPM framework accurately estimates performance and energy consumption for both short and long basic blocks, addressing a critical limitation in existing techniques.

2.4 Transfer Learning and Fine Tuning

The flexibility of the Transformer architecture makes it exceptionally well-suited for transfer learning and fine-tuning [8, 13, 14], enabling adaptation to various tasks with minimal additional training. Transfer learning involves adapting a pre-trained model to a new task, leveraging the knowledge it has already acquired. This approach is particularly beneficial when dealing with limited data for the new task, as the pre-trained model provides a robust starting point.

There are two representative fine-tuning methods: full fine-tuning and adaptor-based fine-tuning. In the full fine-tuning [8], all parameters of the pre-trained model are updated during training on the new task. This method allows the model to adjust to the new data comprehensively, fine-tuning its weights to optimize performance for the specific task. While full fine-tuning is effective, it can be computationally expensive and requires a significant amount of labeled data to avoid overfitting.

Adaptor-based fine-tuning [13, 14], on the other hand, offers a more efficient alternative. Instead of updating all the parameters of the pre-trained model, small task-specific modules called adaptors are added to the network. These adaptors are trained on the new task while keeping the original model parameters mostly frozen. This approach significantly reduces computational costs and the risk of overfitting, as only a small subset of parameters is updated. Adaptor-based fine-tuning is particularly beneficial when working with limited computational resources or small datasets, as it allows the model to adapt to new tasks or different labels efficiently without extensive retraining.

The Transformer architecture's capability to support transfer learning through both full and adaptor-based fine-tuning makes it a versatile tool for various applications. Full fine-tuning provides thorough adaptation at the cost of higher computational demands, while adaptor-based fine-tuning offers a more resource-efficient approach, enabling effective model adaptation even with limited data and computational power. In our work, we explore the potential of both methods to enhance the cross-microarchitecture adaptability of the proposed models, thereby eliminating the need to learn individual per-architecture models from scratch.

3 DeepPM Framework Overview

Figure 3 shows the comprehensive overview of the proposed DeepPM framework, which predicts the clock cycle and energy consumption of every basic block in a host-compiled target application. This capability allows various technologies to access cycle and energy information for basic blocks without executing them on actual hardware or simulators. Additionally, the framework is designed to be effectively used across various processors, different ISAs, and different microarchitectures. The DeepPM framework achieves the prediction of performance or energy consumption for basic blocks through two main phases: *Input Data Generation* and *Learning*.

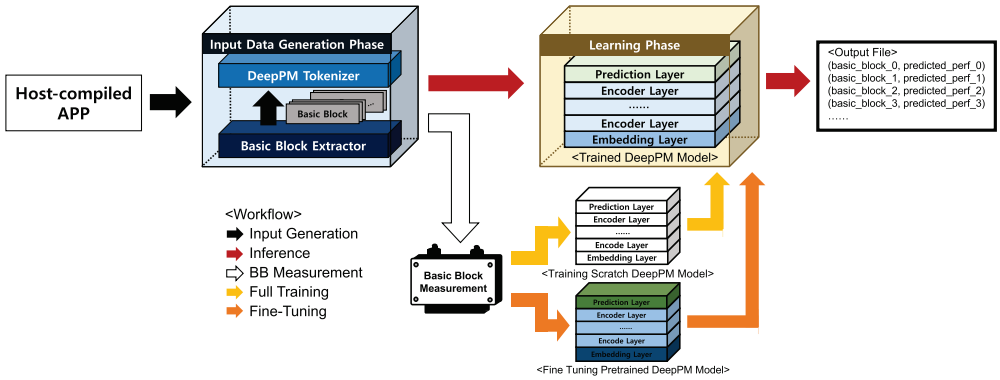


Fig. 3. DeepPM framework overview.

In the Input Data Generation phase, the framework first extracts basic blocks from an ELF-formatted target application using Ghidra [1] within a Python environment. This process verifies whether each function in the target program is included in the .text section and, if so, collects all the basic blocks in both hexadecimal machine code and human-readable assembly format. The DeepPM Tokenizer then processes the extracted basic blocks to generate token vectors. The tokenizer handles basic blocks regardless of the ISA, enabling the DeepPM model architecture to be applied across different processor architectures. Section 4 provides a detailed explanation of the DeepPM Tokenizer.

In the Learning phase, we train the DeepPM model, which is specifically designed to effectively learn the relationship between the instruction sequences of basic blocks and their cycle/energy consumption estimates. Once trained, the model can predict performance or energy consumption from the tokenized basic blocks provided as input. The model can be either fully trained from scratch on a large dataset from the target environment or fine-tuned with a small amount of data from the target environment, having been pretrained in a different environment. Section 5 provides a detailed explanation of the DeepPM model architecture.

4 DeepPM Tokenizer

The DeepPM tokenizer processes input assembly language-based basic blocks, generating tokenized basic blocks for the DeepPM model. To ensure the DeepPM architecture's applicability across various processors, the DeepPM tokenizer provides a universal solution, eliminating the need for ISA-specific tokenizers.

Existing techniques often rely on ISA-specific tokenizers, limiting their broader applicability. Our solution addresses this limitation by creating a universal tokenizer capable of processing instructions from any ISA. The underlying idea is to split an instruction into its operations and operands using space-delimited forms like natural language texts, as this structure is commonly used in most assembly languages.

The tokenization process is performed through the following key steps:

(i) Initial Splitting: Initially, instructions are split into tokens using spaces. This separation typically results in tokens that include opcodes, registers, characters (e.g., '[', ']', ',', '+', '-', '*', and ':'), and numeric values.

(ii) Handling Numeric Values: Numeric values are virtually infinite, unlike opcodes, registers, and characters, which are finite and can be indexed by creating a dictionary during the training

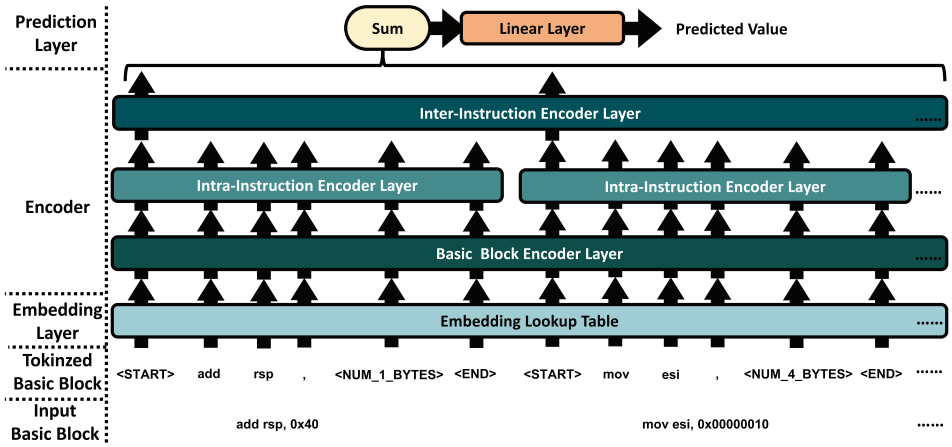


Fig. 4. DeepPM transformer model architecture.

phase. To address this, we use a novel formatting method for numeric value tokens. We form a structure where prefixes and postfixes are determined based on their values. The prefix '<ZERO_' or '<NUM_' is used depending on whether the value is zero or not, respectively, and postfixes such as '1_BYTES>', '2_BYTES>' are determined by the byte size of the value.

(iii) Adding Structural Tokens: Special tokens are added to provide structural markers within both instructions and basic blocks. For instance, '<START>' and '<END>' demarcate the beginning and end of each instruction, while '<BLOCK_START>' and '<BLOCK_END>' delineate the start and end of the basic block.

To better illustrate our tokenization procedure, consider the instruction 'cmp byte ptr [rbp-0x00001049], 0x00'. Initially, it is tokenized based on the space delimiter, resulting in the tokens: 'cmp', 'byte', 'ptr', '[', 'rbp', '-', '0x00001049', ']', ',', and '0x00'. The numeric value '0x00001049' is represented as '<NUM_4_BYTES>' because its value is non-zero and it is 4 bytes in size. On the other hand, '0x00' is represented as '<ZERO_1_BYTES>' because its value is zero and it is 1 byte in size. With the addition of the special tokens, the final result becomes '<START>', 'cmp', 'byte', 'ptr', '[', 'rbp', '-', '<NUM_4_BYTES>', ']', ',', '<ZERO_1_BYTES>', and '<END>', forming the tokenized basic block.

Similar to general NLP practices, the DeepPM tokenizer performs token-to-index mapping through vocabulary creation during the training process. The DeepPM model utilizes these index vectors instead of the tokenized basic blocks themselves, enabling efficient processing and prediction. Based on this universal approach, the DeepPM Tokenizer supports cross-ISA and cross-microarchitecture adaptability, making it a robust solution for diverse processor environments.

5 DeepPM Model Architecture

In this section, we elaborate on the DeepPM model architecture, illustrated in Figure 4. The DeepPM model architecture comprises three main components: the *Embedding Layer*, the *DeepPM Encoder*, and the *Prediction Layer*.

The first component, the Embedding Layer, converts all tokens of the tokenized basic blocks into embedding vectors using an embedding table. This process ensures that each token is represented as a dense vector, capturing its syntactic and semantic properties. The second component, the DeepPM encoder, encodes these embedding vectors. While fundamentally based on the

Transformer encoder architecture [32], the DeepPM encoder introduces three key modifications to address the unique challenges posed by basic blocks, which differ significantly from natural language text. Basic blocks consist of sequences of instructions with specific syntactic and semantic rules that influence performance and energy consumption. Therefore, the key challenge lies in effectively capturing the dependencies and relationships within these sequences. To address these challenges, the DeepPM encoder incorporates a multi-head weighted self-attention mechanism to adjust the influence of tokens based on their positions and interactions within the basic block. It also removes layer normalization to align with our goal of predicting quantitative values like cycle or energy consumption, where absolute values are crucial. Additionally, the encoder introduces three specialized layer types to account for different structural aspects of the basic block. The third component, the Prediction Layer, combines the encoded vectors to generate the desired prediction values, such as cycle or energy consumption. This layer aggregates the encoded vectors and passes them through a linear transformation to produce the final prediction. The following subsections discuss each component in detail.

5.1 Embedding Layer

The Embedding Layer is the first component of the DeepPM model architecture. It takes input from a tokenized basic block generated by the DeepPM Tokenizer. Upon receiving the input, the Embedding Layer looks up an embedding table to convert each token index into a d_{model} dimensional embedding vector.

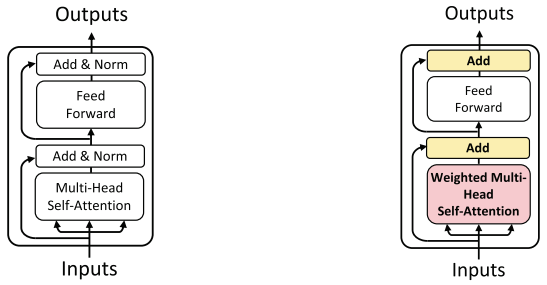
To preserve the sequential information of tokens within the basic block, positional encoding is applied to the embedding vectors. This encoding incorporates sequence order information, allowing the model to understand the relative positions of tokens within the basic block. As a result, the output of the Embedding Layer is a set of embedding vectors that not only represent each token but also retain the structural context of the basic block. This transformation ensures that the DeepPM model has a rich and informative representation of the input data as a dense vector, capturing its syntactic and semantic properties.

5.2 DeepPM Encoder

The DeepPM encoder processes input embedding vectors representing basic blocks to generate context-aware encoded vectors. To improve the learning and representation of basic blocks, we modify three key components of the original Transformer encoder architecture: *weighted attention score, removal of layer normalization*, and *three different types of encoder layers depending on the token types*. It is noteworthy that the first two modifications are implemented within the structure of the encoder layer, while the third modification is applied when stacking the encoder layers within the encoder.

5.2.1 Weighted Attention Score. The core of the Transformer encoder is the attention mechanism, which captures contextual relationships within a sequence. In NLP applications, the typical attention mechanism computes the attention score between tokens using the *Query* and *Key* derived from the embedding vectors representing the tokens. This score calculation, based on the dot product, ensures that tokens with higher similarity have higher attention scores, indicating a stronger relationship or similarity between them. This score is then applied to the *Value* to influence the token representation, allowing every token within the sequence to influence each other, regardless of their distance. This capability enables the model to better understand contexts in NLP tasks and effectively learn relationships between all tokens within the text sequence.

However, unlike in the NLP case, the influence of tokens in the context of basic blocks can vary significantly based on their distance and specific operations. For example, if data is written to the



(a) Original (b) DeepPM

Fig. 5. Original transformer encoder layer architecture and DeepPM encoder layer architecture. DeepPM takes the weighted attention score and removes layer normalization.

'rax' register and then immediately read in the next instruction, the relationship between these two 'rax' tokens is highly significant as they likely pertain to the same data operation. Conversely, if 'rax' is written in one instruction and then read ten instructions later, other instructions may have modified 'rax' in the meantime, making the relationship between these two instances of 'rax' less significant. Additionally, the influence between opcodes can vary depending on their distance from each other, and the relationship between an opcode and its operands can also change based on their distance.

To explicitly account for these aspects in our design, we propose a multi-head weighted self-attention mechanism that incorporates distance-based weighting into the calculation of the attention score. This mechanism ensures that the attention mechanism more accurately reflects the relevance of different tokens based on their positions and roles within the basic block. The attention calculation formula of the DeepPM encoder layer, reflecting these weights, is as follows:

$$Weighted\_Attention(Q, K, V) = softmax\left(\frac{QK^T \circ R}{\sqrt{d_k}}\right)V, \tag{1}$$

Q, K, and V are the matrices of the *Query*, *Key*, and *Value*, respectively; d_k is the dimensionality of the K; and $\circ$ is the element-wise multiplication. R is the distance-weighted ratio matrix, defined as the distance-based weight we applied:

$$R = \begin{bmatrix} a_{11} & \cdots & a_{1s} \\ \vdots & \ddots & \vdots \\ a_{s1} & \cdots & a_{ss} \end{bmatrix}, a_{ij} = \begin{cases} (s+i-j)/s & \text{if} i \leqq j \\ (s-i+j)/s & \text{if} i > j \end{cases}, \tag{2}$$

where s is the length of the input sequence. The encoder layer in the DeepPM model incorporates this multi-head weighted self-attention mechanism, as depicted in Figure 5(b).

5.2.2 Removing Layer Normalization. In the fields of NLP and CV, the relative relationships between vectors often hold more significance than the absolute values of the vectors themselves. Layer normalization is commonly used in these domains to maintain the relative relationships between values while regulating their magnitudes within a consistent range. The original Transformer encoder layer architecture, as depicted in Figure 5(a), also utilizes normalization to achieve these benefits.

However, the primary goal of the DeepPM framework is to predict quantitative values, such as core cycle or energy consumption, rather than to produce typical output vectors. Since the

absolute values of the output vectors are also crucial in our case, the output vectors should deliver appropriate magnitudes directly related to the core cycle or energy consumption. Consequently, the use of layer normalization does not align with the prediction objectives of DeepPM.

Therefore, we remove the layer normalization from the DeepPM encoder layer architecture. By eliminating normalization, we allow the model to retain the absolute values necessary for accurate quantitative predictions. Through our experiments, detailed in Section 7.2.3, we confirmed that omitting layer normalization resulted in improved accuracy for performance and energy consumption predictions. The modified DeepPM encoder layer architecture, as shown in Figure 5(b), does not incorporate normalization, aligning it more closely with the framework's predictive goals.

5.2.3 Structural Encoder Layers. The original Transformer encoder consists of multiple *num_layers* identical layers, each performing calculations across the entire input sequence. These layers iteratively capture contextual relationships within the input vector sequence, generating output vectors that encapsulate the overall semantic meaning of the sequence. This approach enhances the understanding of contextual information and improves feature representation.

In NLP, the focus is often on the relationships between words rather than their individual structures, reflecting the complexity and diversity of natural language. However, basic blocks in computing have a specific structure composed of sequences of instructions, each with its own internal organization. Understanding this structure is crucial for accurately predicting performance and energy consumption, as each instruction plays a distinct role in the execution process.

To address these characteristics, the DeepPM encoder implements a novel approach where each layer processes input vectors based on different structural aspects of the basic block. Depending on these aspects, each encoder layer functions as one of three types: Basic Block Encoder Layer, Intra-Instruction Encoder Layer, or Inter-Instruction Encoder Layer, as shown in Figure 4.

The Basic Block Encoder Layer performs attention calculations among all vectors in the input sequence, similar to the original encoder layer. It aims at achieving a comprehensive understanding across all individual tokens in the basic block. However, it operates based on the modified DeepPM encoder layer structure, incorporating distance-based weighting and the absence of layer normalization as illustrated in Figure 5(b).

Next, the Intra-Instruction Encoder Layer focuses on calculating the relationships between tokens *only within the same instruction*. This is because tokens belonging to the same instruction may have a higher impact on the basic block's overall performance and energy consumption. For example, the cycles and energy required for a 'mov' opcode depend heavily on its operands since they determine whether the data movement is between registers or between a register and memory. Additionally, the role of the same 'rax' register, whether as a source or destination, affects the estimation outcome. Therefore, we have employed the Intra-Instruction Encoder Layer so that it can more focus on learning the relationships among vectors belonging to the same instruction.

Lastly, the Inter-Instruction Encoder Layer calculates the relationships among the vectors representing each instruction within the basic block. It allows for considering the fact that the overall performance and energy consumption depend significantly on the interactions between instructions. This layer uses the start tokens for each instruction, which have been processed and transformed by the previous two layers, to encode the relationships among instructions. Calculating these relationships ensures that the model captures the dependencies and interactions between instructions, essential for understanding the basic block's overall behavior and performance.

Through these three types of layers, the DeepPM encoder captures the complex dependencies within the input basic block and returns vectors corresponding to instructions. The effectiveness of each layer type has been demonstrated through results in Section 7, showing that models utilizing

each individual layer achieve higher accuracy compared to the original Transformer encoder. We observed that the best performance is achieved when all three layers are utilized together, highlighting the importance of capturing both intra- and inter-instruction relationships along with the impact of individual tokens holistically.

5.3 Prediction Layer

The Prediction Layer is the final component of the DeepPM model architecture. It takes the context-aware encoded vectors generated by the DeepPM encoder and produces a single prediction value, such as cycle or energy consumption. As described earlier, the DeepPM encoder processes a sequence of embedding vectors representing a tokenized basic block and outputs vectors corresponding to the individual instructions within the block. Given that the overall cycle or energy consumption of a basic block is influenced by the number of instructions it contains, we need to aggregate all these instruction vectors to capture the cumulative effect.

To achieve this, the encoded vectors are summed to form a single vector that represents the entire basic block. This summed vector encapsulates the aggregated information necessary for the final prediction. This vector is then passed through a linear layer, which transforms it into the final predicted value. The linear layer applies a learned transformation to the summed vector, effectively mapping the aggregated instruction-level information to a quantitative prediction of cycle or energy consumption. This design ensures that the model can accurately predict performance and energy metrics based on the detailed structural and contextual information captured by the DeepPM encoder.

6 Experimental Setup

This section outlines the datasets and introduces the models trained for the experiments. To evaluate the proposed DeepPM framework over different processor environments, we constructed six datasets consisting of basic blocks and their performance or energy consumption labels. These datasets were used for (i) training the DeepPM model from scratch and (ii) fine-tuning to demonstrate the model's capability across various ISAs and microarchitectures. Building on these datasets, we trained and evaluated multiple models to demonstrate the effectiveness of the proposed DeepPM architecture and tokenizer.

For the **full training (FT)** of the DeepPM model architecture, we used four datasets to evaluate the model's ability to predict performance and energy consumption on different ISAs. These datasets were derived from the SPEC CPU 2017 benchmark suite [6], which is widely used for CPU performance evaluation. Two datasets were collected from x86 architectures, and the other two were from ARM architectures. The x86 datasets contain approximately 700,000 basic blocks, while the ARM datasets contain about 400,000 basic blocks. Each architecture has one dataset labeled with performance measurements and another with energy consumption measurements. The x86 performance and energy data were collected from an Intel Coffee Lake CPU, while the ARM data were collected from a Cortex-M4F CPU. The detailed methodologies used to create these datasets, including the measurement tools developed for x86 and ARM architectures to evaluate the performance and energy consumption of basic blocks, are provided in Sections 6.1 and 6.2.

The remaining two datasets were used to demonstrate the applicability of the DeepPM approach through fine-tuning, targeting different microarchitectures within the same ISA. One dataset consists of a small set of x86 basic blocks from applications in the Phoronix Test Suite [21], labeled with cycle data from an Intel Alder Lake CPU, containing approximately 60,000 data points. This dataset illustrates that a pre-trained DeepPM model, initially trained on a large dataset from one microarchitecture (Intel Coffee Lake) with the SPEC CPU 2017 benchmark, can be effectively adapted to another microarchitecture (Intel Alder Lake) through fine-tuning only

with a smaller Phoronix dataset. The other dataset is an FT set of SPEC CPU 2017 basic blocks labeled with performance data from the Alder Lake CPU. This dataset is utilized to quantitatively compare the effectiveness of the fine-tuning methodology against the FT approach. The detailed methodologies for creating these datasets are provided in Section 6.3.

To evaluate the DeepPM architecture and tokenizer, we developed models trained on the provided datasets. For comparison, we created baseline models using the state-of-the-art ML-based basic block learning Ithemal architecture [22]. Since the DeepPM architecture builds upon the Transformer structure with multiple optimizations, additional models were constructed, including the original Transformer architecture and its variants with the proposed enhancements. The structures, hyperparameters, and training policies for these models are detailed in Section 6.4.

6.1 x86 FT Datasets

The x86 FT Datasets are used to train models that predict the performance and energy consumption of x86 basic blocks. These datasets are constructed using basic blocks collected from the SPEC CPU 2017 benchmark. The benchmarks are compiled using the default options of GCC, resulting in approximately 700,000 basic blocks. These basic blocks are extracted using the Input Data Generation method described in Section 3. The performance and energy consumption of these basic blocks are measured on an Intel Coffee Lake CPU, which utilizes the x86 ISA. The performance metric, i.e., clock cycles, is measured using a tool adapted from BHive [4], while energy consumption is measured using a novel tool integrating Running Average Power Limit (RAPL) [5]-based energy measurement with BHive.

6.1.1 x86 Clock Cycle Measurement. The BHive tool [4] provides a method for profiling the throughput of arbitrary basic blocks, which has been initially used by Ithemal [22] to profile cycle counts. The technique involves unrolling each basic block 200 and 100 times, respectively, and executing them to measure the consumed clock cycles. The final cycle count is derived from the difference between the executions performed 200 times and 100 times. This approach mitigates inaccuracies from excessively short runs and aims at excluding influences unrelated to the basic block's performance. While BHive originally measured additional CPU-related information, we modified it to focus solely on clock cycles and adjusted its internal settings to suit our experimental environment.

6.1.2 x86 Energy Consumption Measurement. Intel's RAPL technology [5] monitors the energy consumption of processors. We integrate RAPL's energy monitoring capabilities with the BHive methodology to create a tool for measuring the energy consumption of basic blocks. RAPL has a sampling period, updating its registers every 1 ms [11]. To synchronize with these sampling cycles, we fix the CPU frequency and calculate the minimum unrolling number for each basic block based on measured cycle values and the fixed frequency. We collected the RAPL measurements multiple times to ensure data consistency, verifying that the collected results of energy consumption do not vary significantly. The median of these measurements is used as the final label.

6.2 ARM FT Datasets

The ARM FT Datasets are designed to train models for predicting the performance and energy consumption of ARM basic blocks. We cross-compiled the SPEC CPU 2017 benchmark by utilizing the arm-linux-gnueabihf-gcc and gfortran-arm-linux-gnueabihf. These basic blocks are then extracted using the Input Data Generation method in the similar way used in x86. As a result, we can collect approximately 400,000 basic blocks for the ARM architecture. The clock cycle and energy consumption measurements are conducted on the Cortex-M4F processor using the MAX78000 evaluation kit [15].

To efficiently conduct cycle and energy consumption measurements for each basic block, we developed an automated measurement tool. The tool first generates a C program that includes the target basic block and measurement code. During this process, instructions within the basic blocks that impact the program stack, such as push/pop, or low power mode-related instructions, such as wfi/wfe, are excluded to ensure feasible execution. For the given basic block, the tool generates a C program, compiles it into an ELF file format, and uploads the binary to the board using OpenOCD [24]. Once the measurement is done, the tool receives the results for the program executed on the board via the Minicom interface.

6.2.1 ARM Clock Cycle Measurement. Clock cycle measurement utilizes a 24-bit counter in the ARM processor known as 'systick'. By inserting code to start and stop the 'systick' timer before and after the execution of a basic block, we determine the core cycle difference. Due to the short length of the basic block, a single execution may not accurately update the value of this counter. Thus, we unrolled (i.e., multiplied) the basic block multiple times during the code generation procedure to repeatedly execute it with a sufficiently long execution time.

6.2.2 ARM Energy Consumption Measurement. We measured the energy consumption using the power monitor provided by the MAX78000 board. During this process, we accounted for the board's minimum recommended time interval for power and energy measurement, 100 ms. Considering the operating frequency, we determined the number of unrolls for a basic block to ensure that measurements are taken for the execution running at least 100 ms.

6.3 x86 Fine-Tuning Datasets

The x86 Fine-Tuning Datasets consist of approximately 70,000 basic blocks collected from applications in the Phoronix Test Suite [21]. The extraction of basic blocks and the measurement of their performance are conducted in the same manner as with the x86 FT Datasets.

It is noteworthy that, while the full training datasets are measured on the Intel Coffee Lake CPU, the fine-tuning datasets are measured on the Intel Alder Lake CPU. This allows us to evaluate the fine-tuning capability across different microarchitectures in the same x86 ISA family. Also, the number of basic blocks of the Phoronix Test Suite is significantly smaller than those collected using the SPEC CPU 2017 benchmark, making it suitable for evaluating the adaptability of the DeepPM framework in a fine-tuning scenario with limited target-specific data.

We also collect an FT dataset using SPEC CPU 2017 benchmark on the Intel Alder Lake CPU. We train a model from scratch using this dataset to establish a baseline trained with a sufficiently large dataset. We utilize this baseline to compare it with the fine-tuned model only with the small Phoronix dataset. This comparison allows us to quantitatively evaluate how the DeepPM framework can be adapted to new microarchitectures with minimal additional data.

6.4 Model Configurations and Training Policies

Table 1 summarizes the trained models evaluated in this section. DeepPM refers to the model based on the DeepPM architecture proposed in Section 5, while Ithemal refers to the model using the Ithemal architecture, and Trans refers to the model based on the original Transformer encoder architecture. Models indicated with a "+", such as DeepPM+, use the DeepPM tokenizer proposed in Section 4; in contrast, models without the "+" use the Ithemal tokenizer [22]. The Transformer encoder structure includes four configurations. "Original" in Transformer Encoder refers to the original encoder of the Transformer architecture [32], while "WA" represents the Weighted Attention Score as discussed in Section 5.2.1. "RN" refers to the technique removing layer normalization described in Section 5.2.2, and "3E" denotes the three-types encoder layers as explained in Section 5.2.3.

Table 1. The Components of Each Model Architecture

Model Name	LSTM	Transformer Encoder				DeepPM Tokenizer
		Original	WA	RN	3E	
DeepPM+	-		o	o	o	o
DeepPM	-		o	o	o	
Ithemal+	o	-	-	-	-	o
Ithemal	o	-	-	-	-	
Trans+	-	o				o
Trans_WA+	-		o			o
Trans_RN+	-			o		o
Trasns_3E+	-				o	o
Trans	-	o				
Trans_WA	-		o			
Trans_RN	-			o		
Trasns_3E	-				o	

Throughout all model configurations, the number of encoder layers is consistently set to eight. While all other versions use identical layers, the "3E" configuration maintains a total of eight layers but comprises different types: it includes two Basic Block Encoder Layers, two Intra-Instruction Encoder Layers, and four Inter-Instruction Encoder Layers.

For evaluating each model, we divide each dataset into training, validation, and test datasets at an 8:1:1 ratio. The training dataset is used for model training, the validation dataset is employed to validate the results of each epoch and select the best model, and the test dataset is utilized to compare the selected best model with other architectures. Hyperparameters are adjusted based on validation results, and final evaluations are performed on the test set.

All models undergo FT for 30 epochs and finetuning for 10 epochs, using the L1-based loss function. The Ithemal architecture-based models are trained using dimensions of 256, a batch size of 4, SGD for the optimizer, and an initial learning rate of 0.5. After the first two epochs, the learning rate decays by a factor of 1.2. These settings are mostly the same as those used in [22], except for the initial learning rate. The DeepPM architecture-based models utilize dimensions of 512, a batch size of 4, the Adam optimizer, and an initial learning rate of 0.00001, which is managed by a LinearLR scheduler.

7 Evaluation Results

To validate the effectiveness of the proposed techniques, we train various models and conduct comprehensive comparisons. This section presents the experimental results in three main areas.

First, we compare our proposed DeepPM architecture-based models with the Ithemal architecture-based models, which represent the state-of-the-art basic block learning approach [22], focusing on actual program execution scenarios. Second, we analyze and compare test dataset results from x86 and ARM FT Datasets, demonstrating the specific advantages and effectiveness of DeepPM architecture and tokenizer. Third, we focus on fine-tuning DeepPM on different microarchitectures within the same ISA using the x86 Fine-Tuning Datasets.

7.1 Prediction Quality Evaluation During Entire Application Executions

We first evaluate the effectiveness of our proposed DeepPM architecture, concentrating on scenarios reflective of actual program executions. In this experiment, we compare our main model, DeepPM+, with the baseline Ithemal model using the x86 FT dataset. As noted in Section 2.2, the

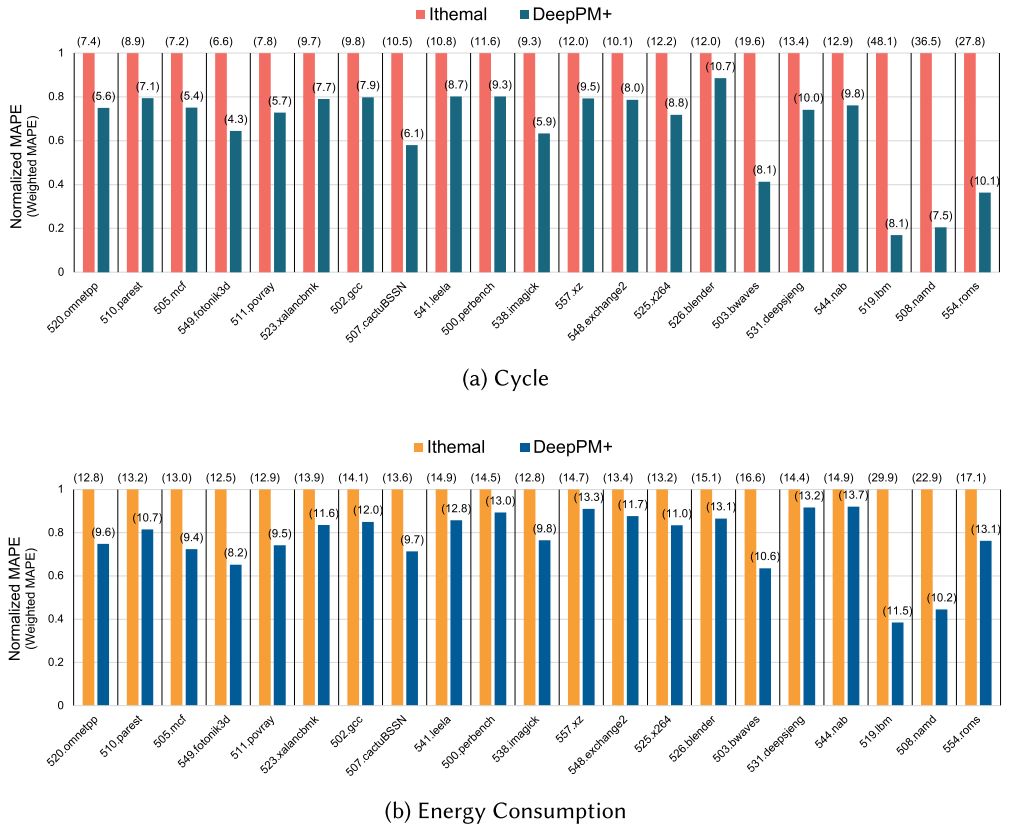


Fig. 6. Normalized weighted MAPE of basic blocks during application execution. The values in parentheses indicate the weighted MAPE values before normalization.

Ithemal model struggles to predict long basic blocks accurately, despite long basic blocks significant contribution to overall execution in many applications. The performance of both models is evaluated by considering the proportions of basic block lengths observed in actual program executions, demonstrating that DeepPM achieves better predictive accuracy by addressing these limitations.

For this evaluation, we define a metric termed ***Weighted Mean Absolute Percentage Error (MAPE)*** to accurately reflect the contribution of each basic block to the entire application execution. This metric accounts for the appearance frequencies of basic blocks encountered during actual execution in the x86 environment. The weighted MAPE was calculated using the test dataset results from the models that are trained with the x86 FT Datasets. Initially, we calculated the average MAPE for each instruction count by averaging the MAPEs of all basic blocks with the same instruction count. Also, for each application, we identified the frequency of each instruction count for the executed basic blocks. These frequencies were then used to compute the weighted average MAPE by multiplying the MAPE of each instruction count by its corresponding frequency and then summing these values.

Figure 6 illustrates the weighted MAPE for applications in the SPEC CPU 2017 benchmark, normalized to the baseline. The results indicate that DeepPM+ consistently outperforms Ithemal across all experimental results. Specifically, we observe an average improvement of 34% in cycle

prediction and 23% in energy consumption prediction. Notably, as depicted in Figure 1, applications that utilize longer basic blocks exhibit substantial performance improvements, with increases of up to 83% in cycle prediction efficiency and up to 62% in energy consumption prediction.

The evaluation results confirm that DeepPM+ significantly addresses the limitations of the baseline Ithemal model, as demonstrated by its enhanced accuracy when accounting for the proportions of basic block lengths observed in actual program executions. Notably, the performance improvements are particularly pronounced for applications with longer basic blocks, achieving substantial gains in both cycle and energy consumption predictions. These results highlight the practical advantages of DeepPM+ in addressing key challenges in performance prediction tasks, offering a more reliable and versatile solution compared to the state-of-the-art approach.

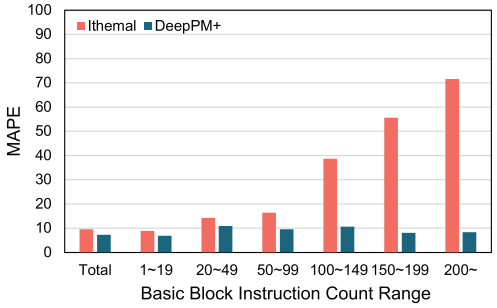
7.2 In-Depth Analysis of Prediction Results

To better understand the highly accurate prediction results of the proposed DeepPM, we delve deeper into analysis to substantiate the efficacy of our architecture. In this section, we compare the test dataset results of Ithemal and DeepPM+ across the x86 FT Datasets, as well as Ithemal+ and DeepPM+ across the ARM FT Datasets. Given that the Ithemal tokenizer only supports the tokenization of x86 basic blocks, we use Ithemal+ as a baseline for the ARM environment by combining the proposed DeepPM tokenizer with the Ithemal model architecture. As mentioned in Section 2, existing ML-based basic block prediction approaches not only struggle with long basic blocks but also face general applicability across processors. The analysis shows that DeepPM achieves strong performance both overall and across various basic block length ranges, while also demonstrating the effectiveness of the DeepPM tokenizer in diverse ISAs. Additionally, test results from various Transformer-based intermediate models are also presented, demonstrating the performance contributions of the three key modifications proposed in this work. Finally, by comparing the training time, inference time, and model size of DeepPM and Ithemal, and conducting comparisons at equivalent model sizes, we substantiate the impact and effectiveness of our contributions.

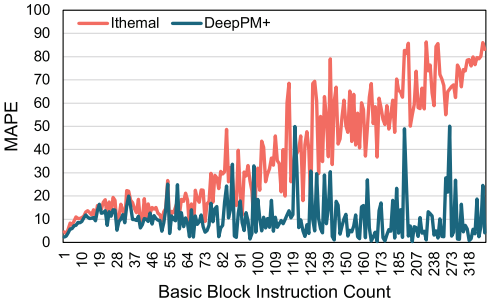
7.2.1 Models Prediction Quality vs. Basic Block Length. As discussed in Section 2.2, a significant limitation of existing techniques is their inability to accurately estimate performance and energy consumption for long instruction sequences. In contrast, the proposed DeepPM framework employs a newly devised transformer architecture capable of effectively handling both long and short basic blocks.

To provide a detailed comparison, Figure 7 presents the MAPE results for cycle and energy consumption predictions using the x86 FT Datasets. As shown in Figure 7(a), the overall average MAPE values for cycle predictions are 9.5% for Ithemal and 7.24% for DeepPM+, marking an improvement of approximately 24%. Notably, when basic blocks are categorized by the number of instructions, the performance disparity between Ithemal and DeepPM+ becomes increasingly pronounced. For the smallest instruction count group (1–19), Ithemal and DeepPM+ achieve MAPE values of 8.82% and 6.89%, respectively, achieving an improvement of about 21%. As the instruction count increases, the performance gap widens, with the largest group (200- instructions) showing MAPE values of 71.63% for Ithemal and 8.32% for DeepPM+, improving the accuracy by 88%. This trend is further presented in Figure 7(b), where the average MAPE for each instruction count is analyzed. These results corroborate that the difference in prediction accuracy between Ithemal and DeepPM+ becomes more pronounced as the instruction count increases.

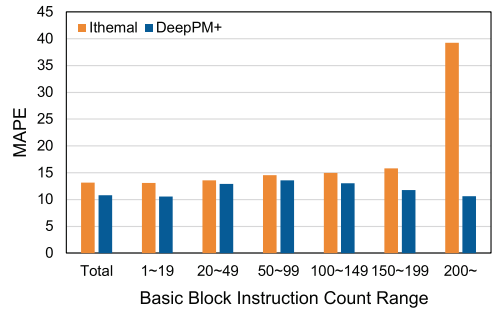
Figure 7(c) and (d) shows the MAPE results for energy consumption predictions from the x86 FT Dataset. While the improvements are not as substantial as for cycle predictions, we observed a similar trend. The average MAPE values are 13.18% for Ithemal and 10.79% for DeepPM+, indicating an improvement of 18%. For the largest instruction count group (200- instructions), MAPE values are 39.25% for Ithemal and 10.61% for DeepPM+, demonstrating an improvement of 73%.



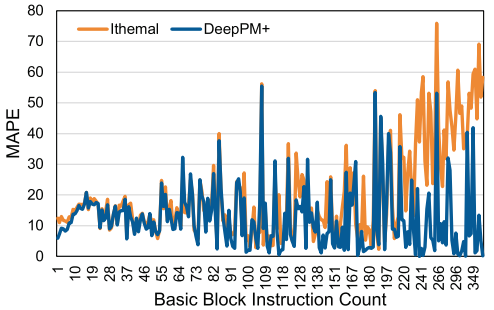
(a) Per Basic Block Group, Cycle Dataset



(b) Per Instruction Count, Cycle Dataset



(c) Per Basic Block Group, Energy Dataset



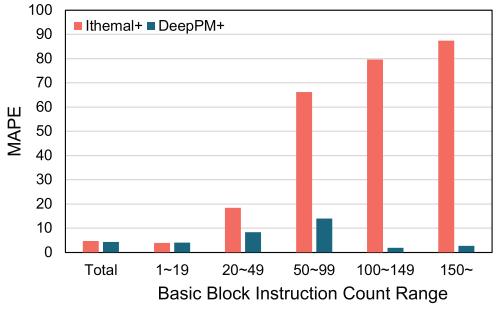
(d) Per Instruction Count, Energy Dataset

Fig. 7. Average MAPE of x86 cycle and energy consumption datasets.

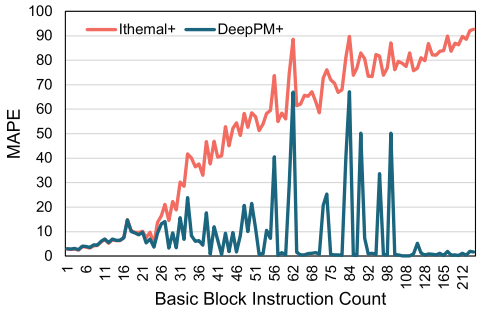
To validate the model performance across different processor architectures, we analyzed the MAPE results for the ARM FT Datasets. As shown in Figure 8, for the cycle estimation, Ithemal+ achieves an overall MAPE of 4.7%, while DeepPM+ achieves an overall MAPE of 4.27%, resulting in an improvement of 9% for DeepPM+. The maximum improvement observed is 97% in the largest group (150- instructions), with the Ithemal+ model achieving 87.37% and the DeepPM+ model achieving 2.63%. For the energy estimation, Ithemal+ achieves an overall MAPE of 5.57%, while DeepPM+ achieves an overall MAPE of 5.42%, resulting in an improvement of 3% for DeepPM+. The maximum improvement observed in the largest group (150- instructions) is 95%, with Ithemal+ achieving 78.08% and DeepPM+ achieving 3.81%.

These results show that DeepPM+ outperforms Ithemal in predicting both cycle and energy consumption across different instruction counts and processor architectures. Its strength is particularly evident for longer basic blocks, where DeepPM+ effectively overcomes a key limitation of the previous approach. As highlighted in Section 7.1, this advantage translates into better alignment with real application execution scenarios, emphasizing the practical impact of these improvements. Moreover, the DeepPM tokenizer is shown to work effectively across multiple ISAs, resolving the challenges posed by the ISA-specific tokenization processes of prior methods.

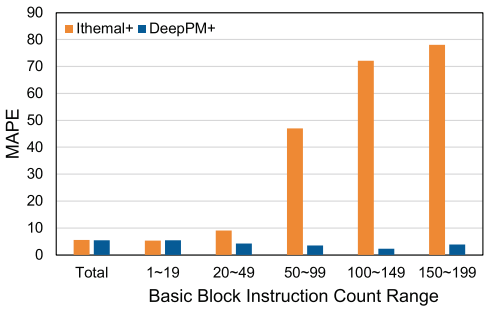
7.2.2 Models Prediction vs. Measured. To further understand the difference in performance, we analyze the relationship between measured and predicted values using heatmaps. Figures 9 and 10 illustrate these heatmaps for the x86 and ARM FT datasets. The heatmaps have the y-axis denoting the measured values and the x-axis denoting the predicted values, with darker colors indicating regions of higher data density. The Cycle Datasets use clock cycles as the unit,



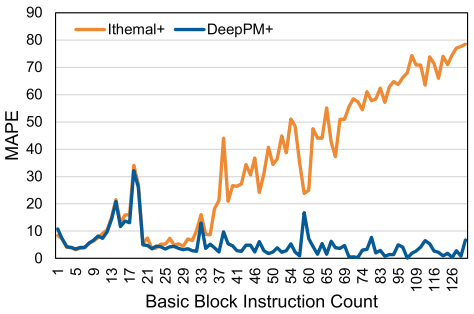
(a) Per Basic Block Group, Cycle Dataset



(b) Per Instruction Count, Cycle Dataset



(c) Per Basic Block Group, Energy Dataset



(d) Per Instruction Count, Energy Dataset

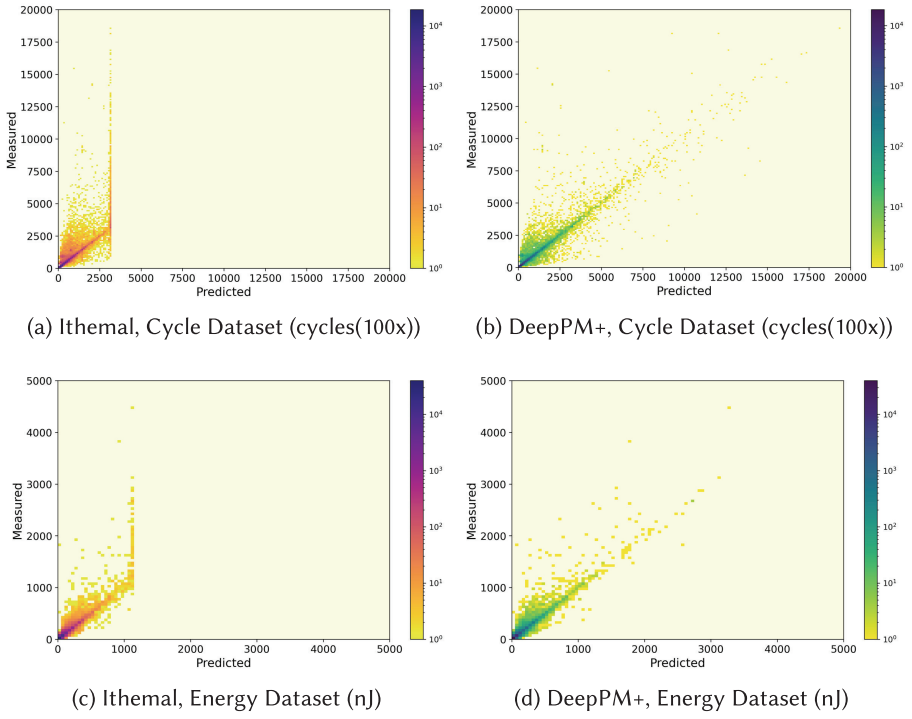
Fig. 8. Average MAPE of ARM cycle and energy consumption datasets.

while the Energy Datasets use nJ (nanoJule). The Ithemal and Ithemal+ models, utilizing the LSTM-based Ithemal architecture, tend to converge to a specific prediction level, regardless of the dataset or the target metric. This convergence often results in poor predictions for longer basic blocks, limiting their effectiveness. Conversely, the DeepPM+ models, based on our proposed DeepPM architecture, maintain a more consistent diagonal pattern in the heatmaps, indicating a stable relationship between measured and predicted values across different environments and prediction targets. This consistency demonstrates the robustness of the DeepPM architecture in effectively handling basic blocks of various lengths across various processors.

7.2.3 Effects of DeepPM Tokenizer and Encoder Architecture. We further compare various models, as listed in Table 1, to demonstrate the effectiveness of our proposed DeepPM tokenizer and encoder structure. Figure 11 shows the evaluation results based on the x86 FT dataset. The performance of DeepPM+ is set as the baseline (normalized to 1), with other models' performances normalized accordingly.

The results show that models utilizing the DeepPM tokenizer (denoted with +) consistently outperform those using the Ithemal tokenizer. DeepPM+ shows an improvement of approximately 17% over DeepPM, Ithemal+ shows an improvement of about 15% over Ithemal, and Trans+ shows an improvement of around 14% over Trans. Overall, there is an average improvement of 15% across the different models.

To validate the three key modifications of the DeepPM model described in Section 5.2, we evaluate various models based on the Transformer architecture (Trans and Trans+). When combining these three techniques, DeepPM+ and DeepPM show performance improvements of approximately



(a) Ithemal, Cycle Dataset (cycles(100x)) (b) DeepPM+, Cycle Dataset (cycles(100x))

(c) Ithemal, Energy Dataset (nJ) (d) DeepPM+, Energy Dataset (nJ)

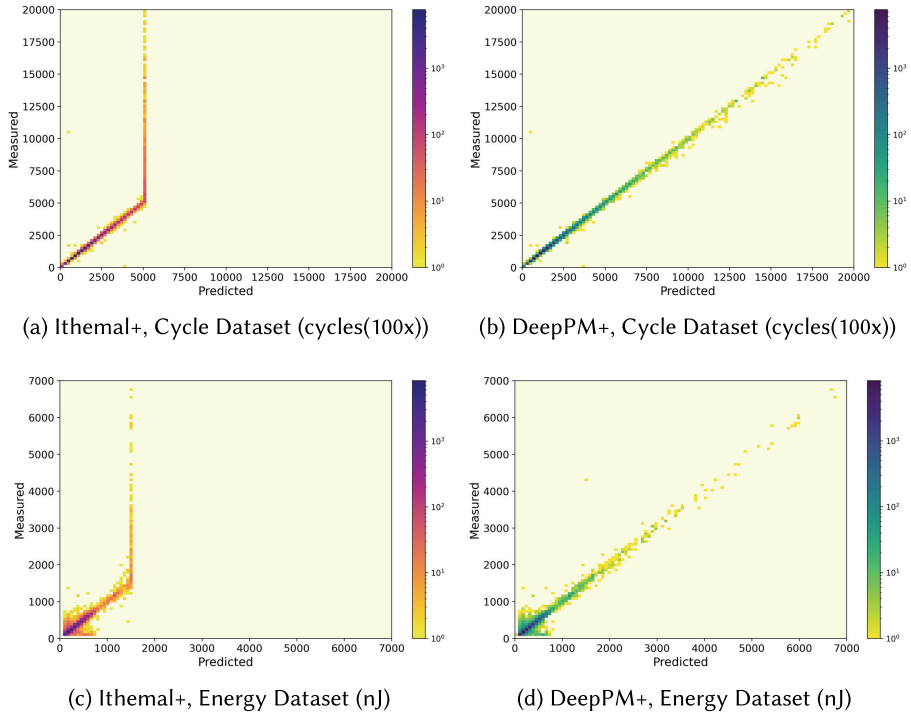
Fig. 9. Heatmaps of measured and predicted values of x86 dataset.

12% and 9% over Trans+ and Trans, respectively. Analyzing each technique individually, using the WA results in performance improvements of approximately 8% and 6% over Trans+ and Trans, respectively. Removing the normalization layer (RN) results in improvements of about 1% each. The three types of encoder layers newly deployed in the DeepPM model achieve improvements of approximately 9% and 7%, respectively.

These results demonstrate the significant contributions of the proposed DeepPM tokenizer and encoder structure to the overall performance improvements of DeepPM. Models utilizing the DeepPM tokenizer consistently outperform those with the Ithemal tokenizer, validating the effectiveness of the DeepPM tokenizer. Additionally, the detailed evaluation of the key modifications in the DeepPM encoder architecture underscores their importance in driving these improvements. Each component plays a distinct role in optimizing model performance, with their combined effects resulting in substantial gains. This analysis confirms that the proposed techniques in Section 5 are critical to achieving the superior predictive capabilities of DeepPM+.

7.2.4 Comparison of Resource Usage. As the DeepPM utilizes the Transformer architecture, it demands more memory resources and computational time compared to the LSTM-based Ithemal architecture when similar dimensions are used. Our results have demonstrated that this increased resource and time investment does not impact our primary contribution: the effective handling of long basic blocks.

As detailed in Section 6.4, Ithemal utilizes a dimensionality of 256, while DeepPM+ operates at a dimensionality of 512. Figure 12(a) illustrates the execution time and model size for both Ithemal and DeepPM+ under these configurations. This figure reveals a significant difference in model size, with Ithemal at approximately 1.2 million parameters and DeepPM+ at around 25.6 million,



(a) Ithemal+, Cycle Dataset (cycles(100x)) (b) DeepPM+, Cycle Dataset (cycles(100x))

(c) Ithemal+, Energy Dataset (nJ) (d) DeepPM+, Energy Dataset (nJ)

Fig. 10. Heatmaps of measured and predicted values of the ARM dataset.

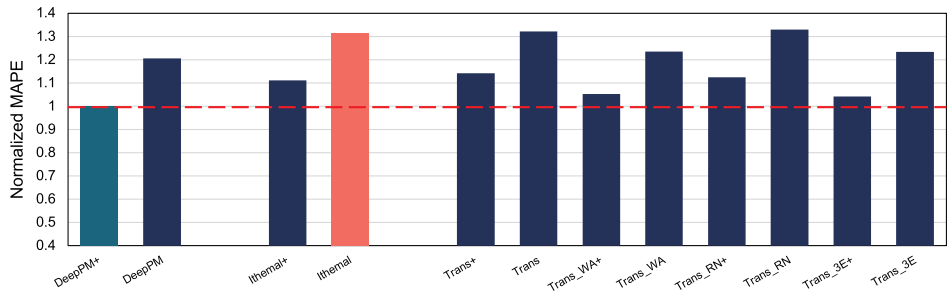
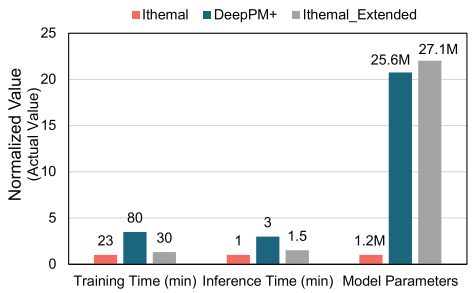


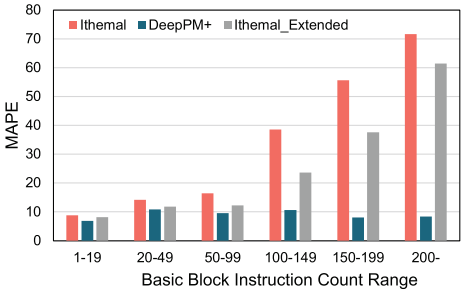
Fig. 11. Test results of various models trained on the x86 FT dataset, demonstrating the effectiveness of our DeepPM tokenizer and encoder structure. Results are normalized to DeepPM+ (baseline).

indicating a substantial increase in model size. Correspondingly, both training and inference times for DeepPM+ are notably higher due to its larger model size.

However, it is crucial to recognize that even when the dimensionality of the Ithemal model is increased to enhance its size, it still does not perform effectively in predicting long basic blocks. As illustrated in Figure 12(a), we developed an Ithemal_Extended model based on the Ithemal architecture with a dimensionality of 1,280, which matches the parameter count of DeepPM+ at approximately 27.1 million. This model was trained on the x86 FT Data's cycle dataset and subsequently tested to evaluate its performance. Figure 12(b) presents the results from this experiment alongside the previously shown average MAPE per basic block group results for Ithemal and DeepPM+ from Figure 7(a). The results, as shown in Figure 12(b), clearly demonstrate that even when the



(a) Model Execution Time and Size



(b) Average MAPE Per Basic Block Group

Fig. 12. Model execution time, size, and average MAPE of x86 cycle.

Ithemal architecture utilizes a similar number of parameters to the DeepPM architecture, it fails to effectively address the core limitation of predicting long basic blocks. This highlights the structural advantages introduced by DeepPM, which are specifically designed to overcome these challenges.

7.3 Fine-Tuning Result

Adapting processors that use the same ISA to different microarchitectures often results in varied performance and energy consumption characteristics. Existing prediction approaches typically focus on a single type of processor, requiring users to relabel basic blocks and perform FT when adapting models to new environments, necessitating time-consuming and resource-intensive processes.

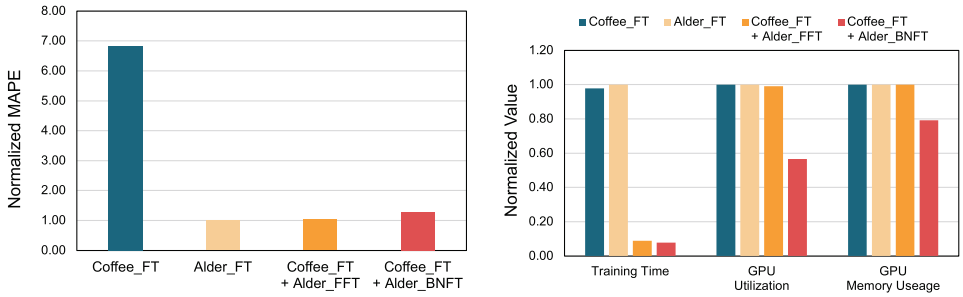
To evaluate how the DeepPM model can mitigate such issues using the fine-tuning process for different microarchitectures, we conducted a comparison experiment involving four different scenarios of the model training, as shown in Figure 13.

These models include two FT models and two fine-tuning models:

(1) **Coffee_FT:** The DeepPM+ model trained on the x86 FT dataset, consisting of SPEC CPU 2017 basic blocks measured on the Intel Coffee Lake CPU, serving as the foundation model.
(2) **Alder_FT:** The DeepPM+ model trained on SPEC CPU 2017 basic blocks measured on the Intel Alder Lake CPU, representing an FT setup for comparison.
(3) **Coffee_FT + Alder_FFT:** The Coffee_FT model fine-tuned using **full fine-tuning** (**FFT**)[8] on the x86 Fine Tuning Dataset, consisting of Phoronix basic blocks measured on the Intel Alder Lake CPU.
(4) **Coffee_FT + Alder_BNFT:** The Coffee_FT model fine-tuned using **bottleneck adaptor fine-tuning** (**BNFT**) [13], which adds a bottleneck adaptor to the model and fine-tunes only this adaptor.

Figure 13(a) shows the average MAPE results for each model on the x86 Fine Tuning Dataset's test data, normalized to the performance of the Alder_FT model. As expected, the Coffee_FT model, fully trained on Intel Coffee Lake data, performed poorly on Intel Alder Lake test data due to the difference in the microarchitecture. In contrast, the Alder_FT model, trained with the Intel Alder Lake data obtained on the target Alder Lake microarchitecture, achieved 13.3%, consistent with trends observed in previous sections.

We observed that the issue of microarchitecture inconsistency in model usage could be mitigated with fine-tuning approaches. Specifically, we found that the two fine-tuning models demonstrated highly accurate performance, comparable to the Alder_FT model. For instance, the Coffee_FT + Alder_FFT model, which modifies a large number of parameters through full fine-tuning,



(a) Intel Adler Lake Test Data Result (b) Training Metrics of Each Model

Fig. 13. Test results and training metrics of fine-tuning for different microarchitectures.

achieved a MAPE of 13.7%, nearly identical to the fully trained Alder_FT model. The Coffee_FT + Alder_BNFT model, which updates fewer parameters by fine-tuning only the bottleneck adaptor, achieved a score of 16.9%. Despite its slightly lower performance, this result confirms the viability of using adaptor-based fine-tuning.

These findings indicate that users can effectively fine-tune a foundation model trained with the DeepPM architecture using a small amount of target environment data to achieve competitive performance without FT from scratch. Furthermore, the DeepPM architecture provides flexible options for users. For instance, when sufficient resources for training computation are available, full fine-tuning can create an effective model in a short time with a small dataset. Conversely, for environments with limited resources, techniques like bottleneck adaptors can be utilized to create an effective model with resource efficiency by updating fewer parameters.

8 Discussion

In this work, we present models trained using our DeepPM techniques. Despite being trained with a smaller dataset compared to previous work [22], our models have demonstrated promising results. We anticipate that training our DeepPM architecture with larger datasets in the future will further enhance predictive performance, leading to even more accurate and reliable models. The main contributions of this work lie in the innovative model architecture, tokenizer, and fine-tuning techniques, rather than the specific datasets or trained models themselves. Therefore, while absolute differences in experimental results may emerge with larger datasets, the overall trends and insights are expected to remain robust and consistent.

The models presented in this study leverage the SPEC CPU 2017 benchmark,which covers a diverse range of application domains [25]. Nevertheless, as with any typical machine learning model, our trained model might exhibit lower performance in domains not represented in this dataset. However, it is important to note that our primary objective extends beyond simply demonstrating a trained model; we aim at introducing a model architecture, tokenization strategies, and fine-tuning techniques. We have demonstrated that these contributions are suitable and effective for basic block learning, and, given the substantial evidence from both our research and prior works supporting the feasibility of basic block learning, we believe that if a specific domain requires a tailored model, users in other fields can readily adapt our approach to develop suitable models for their needs.

Recent advances in NLP research emphasize the importance of data quality over mere quantity to improve model prediction accuracy [3]. In the context of machine learning-based prediction techniques for basic blocks, creating datasets typically involves measuring the cycle count or

energy consumption of each basic block using an unrolling approach. While this method is a common standard [4], it is not without its imperfections. For instance, the unrolling approach often fails to account for key factors that influence real-world execution environments. Pipeline behavior discrepancy arises because basic blocks are executed in isolation during unrolling, which does not reflect the interleaved and dependent nature of instructions in realistic program execution. Similarly, Cache Behavior is overly simplified; unrolling generally assumes an ideal cache state, resulting in an unrealistically high number of L1 cache hits and bypassing effects such as cache conflicts or prefetching mechanisms present in actual scenarios. Branch Prediction is another critical factor, as isolated execution lacks the broader control flow context necessary for accurate branch prediction, unlike real-world environments where predictions rely on global program behavior. While the datasets generated through unrolling techniques have certain limitations compared to real-world execution, they remain highly valuable for predictive modeling. Future advancements in measurement techniques are expected to further enhance the predictive performance of these models.

While basic block-level techniques have proven effective, exploring larger abstraction units such as traces, phases, or functions offers the potential to address some of their inherent limitations. These larger units account for more realistic execution behaviors, including pipeline dynamics and cache interactions, making them better aligned with actual usage scenarios. Given DeepPM's demonstrated strengths in handling long instruction sequences, we anticipate that its architecture and techniques can be effectively extended to these larger abstraction units, further enhancing its applicability to real-world performance optimization tasks.

In summary, our findings highlight the potential of the DeepPM architecture to address performance and energy consumption prediction challenges across different microarchitectures. The flexibility offered by the DeepPM model through techniques such as full fine-tuning and bottleneck adaptors provides practical solutions for both resource-rich and resource-constrained environments. Future work will focus on leveraging larger and more accurate datasets, exploring real-world applications, and extending our techniques to higher-level abstraction units, thereby broadening the impact and applicability of our research.

9 Conclusions

This article introduced the DeepPM framework, leveraging the Transformer architecture to predict performance and energy consumption at the basic block level from compiled binaries, eliminating the need for explicit measurement processes. Additionally, by utilizing the DeepPM tokenizer and fine-tuning techniques, we demonstrate that our approach can be effectively applied across different ISAs and microarchitectures without the need for specific implementations, unlike state-of-the-art ML-based techniques which are typically restricted to a single processor. Our results using the SPEC2017 CPU benchmark suite show that DeepPM achieves lower prediction errors compared to existing models, with a 24% improvement in performance and 18% in energy consumption for x86 basic blocks, and similar gains for ARM processors. Fine-tuning with minimal data further confirmed DeepPM's adaptability, achieving a prediction error of about 13.7%, which is nearly as accurate as the 13.3% error of the fully trained model. These findings highlight DeepPM's capability to significantly enhance the accuracy and efficiency of performance and energy consumption predictions, making it a valuable tool for optimizing computing systems across diverse hardware environments.

The current version of DeepPM can be further improved in several ways. For instance, while DeepPM can predict performance and energy consumption of program execution paths by concatenating the results of individual basic blocks, it could be more efficient if inter-block effects were better modeled. Additionally, we are interested in applying the DeepPM architecture to estimate other design requirements, such as the memory usage of embedded programs.

Acknowledgments

We thank the anonymous reviewers for their valuable comments that greatly improved our article. Special thanks go to Mr. Gyujin Kim for his valuable assistance in developing the DeepPM framework.

References

[1] National Security Agency. 2023. Ghidra. GitHub. Retrieved March 31, 2025 from https://github.com/NationalSecurityAgency/ghidra

[2] Faisal Alam, Preeti Ranjan Panda, Nikhil Tripathi, Namita Sharma, and Sanjiv Narayan. 2014. Energy optimization in android applications through wakelock placement. In *Proceedings of the 2014 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1–4.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[4] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sỳkora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 167–177.

[5] Intel Corporation. 2024. *Intel 64 and IA-32 Architectures Software Developer's Manuals*. Retrieved March 31, 2025 from https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[6] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017 Benchmark. Retrieved March 31, 2025 from https://www.spec.org/cpu2017/

[7] Waltenegus Dargie. 2014. A stochastic model for estimating the power consumption of a processor. *IEEE Transactions on Computers* 64, 5 (2014), 1311–1322.

[8] 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, Volume 1 (Long and Short Papers)*, 4171–4186.

[9] Antonio F. Díaz, Beatriz Prieto, Juan José Escobar, and Thomas Lampert. 2024. Vampire: A smart energy meter for synchronous monitoring in a distributed computer system. *Journal of Parallel and Distributed Computing* 184 (2024), 104794.

[10] Peter Yusuf Dibal, Elizabeth N. Onwuka, Suleiman Zubair, Emmanuel I. Nwankwo, S. A. Okoh, Bala Alhaji Salihu, and H. B. Mustaphab. 2023. Processor power and energy consumption estimation techniques in IoT applications: A review. *Internet of Things* 21 (2023), 100655.

[11] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review* 40, 3 (2012), 13–17.

[12] Timo Hönig, Benedict Herzog, and Wolfgang Schröder-Preikschat. 2019. Energy-demand estimation of embedded devices using deep artificial neural networks. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (PAC)*. 617–624.

[13] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning (PMLR)*. PMLR, 2790–2799.

[14] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, and others. 2022. LoRA: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[15] Maxim Integrated. 2021. *MAX78000*. Retrieved March 31, 2025 from https://www.maximintegrated.com/en/products/microcontrollers/MAX78000.html

[16] Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Wu Ye. 2000. Influence of compiler optimizations on system power. In *Proceedings of the 37th Annual Design Automation Conference (DAC)*. 304–307.

[17] Dongwook Lee, Lizy K. John, and Andreas Gerstlauer. 2015. Dynamic power and performance back-annotation for fast and accurate functional hardware simulation. In *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1126–1131.

[18] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480.

[19] Massinissa Lounis, Ahcène Bounceur, Reinhardt Euler, and Bernard Pottier. 2024. Estimation of energy consumption through parallel computing in wireless sensor networks. *Journal of Ambient Intelligence and Humanized Computing* 15, 2 (2024), 1–13.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (2005), 190–200.
[21] Phoronix Media. 2022. Phoronix Test Suite. Retrieved March 31, 2025 from https://www.phoronix-test-suite.com/
[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 *Proceedings of the International Conference on Machine Learning (PMLR)*. PMLR, 4505–4515.
[23] Rajeev Muralidhar, Renata Borovica-Gajic, and Rajkumar Buyya. 2022. Energy efficient computing systems: Architectures, abstractions and modeling to techniques and standards. *ACM Computing Surveys* 54, 11s (2022), 1–37.
[24] OpenOCD. 2023. Open On-Chip Debugger. Retrieved March 31, 2025 from https://openocd.org/
[25] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. 2018. Wait of a decade: Did spec cpu 2017 broaden the performance horizon?. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 271–282.
[26] Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak. 2000. Function-level power estimation methodology for microprocessors. In *Proceedings of the 37th Annual Design Automation Conference (DAC)*. 810–813.
[27] Santhosh Kumar Rethinagiri, Oscar Palomar, Rabie Ben Atitallah, Smail Niar, Osman Unsal, and Adrian Cristal Kestelman. 2014. System-level power estimation tool for embedded processor based platforms. In *Proceedings of the 6th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*. 1–8.
[28] Norbert Schmitt, Supriya Kamthania, Nishant Rawtani, Luis Mendoza, Klaus-Dieter Lange, and Samuel Kounev. 2021. Energy-efficiency comparison of common sorting algorithms. In *Proceedings of the 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.
[29] Dongkun Shin, Jihong Kim, and Seongsoo Lee. 2001. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers* 18, 2 (2001), 20–30.
[30] Karan Singh, Major Bhadauria, and Sally A. McKee. 2009. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News* 37, 2 (2009), 46–55.
[31] Nima TaheriNejad. 2024. In-memory computing: Global energy consumption, carbon footprint, technology, and products status quo. In *Proceedings of the 2024 IEEE 24th International Conference on Nanotechnology (NANO)*. IEEE, 495–500.
[32] 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 *Proceedings of the Advances in Neural Information Processing Systems*. 5998–6008.
[33] Zhonglei Wang and Jörg Henkel. 2012. Accurate source-level simulation of embedded software with respect to compiler optimizations. In *Proceedings of the 2012 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 382–387.
[34] Zhiyao Xie, Xiaoqing Xu, Matt Walker, Joshua Knebel, Kumaraguru Palaniswamy, Nicolas Hebert, Jiang Hu, Huanrui Yang, Yiran Chen, and Shidhartha Das. 2021. APOLLO: An automated power modeling framework for runtime power introspection in high-volume commercial microprocessors. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14.
[35] Zhuoran Zhao, Andreas Gerstlauer, and Lizy K. John. 2016. Source-level performance, energy, reliability, power and thermal (PERPT) simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 2 (2016), 299–312.

Received 1 August 2024; revised 7 March 2025; accepted 10 March 2025