DeepPM: Transformer-based Power and Performance Prediction for Energy-Aware Software

Jun S. Shim*, Bogyeong Han*, Yeseong Kim†, and Jihong Kim*
*Department of Computer Science and Engineering, Seoul National University
†Department of Information and Communication Engineering, DGIST
Email: *{jsshim, bkhan, jihong}@davinci.snu.ac.kr, †yeseongkim@dgist.ac.kr

***Abstract*—Many system-level management and optimization techniques need accurate estimates of power consumption and performance. Earlier research has proposed many high-level/source-level estimation modeling works, particularly for basic blocks. However, most of them still need to execute the target software at least once on a fine-grained simulator or real hardware to extract required features. This paper proposes a performance/power prediction framework, called Deep Power Meter (DeepPM), which estimates them accurately only using the compiled binary. Inspired by the deep learning techniques in natural language processing, we convert the program instructions in the form of vectors and predict the average power and performance of basic blocks based on a transformer model. In addition, unlike existing works based on a Long Short-Term Memory (LSTM) model structure, which only works for basic blocks with a small number of instructions, DeepPM provides highly accurate results for long basic blocks, which takes the majority of the execution time for actual application runs. In our evaluation conducted with SPEC2006 benchmark suite, we show that DeepPM can provide accurate prediction for performance and power consumption with 10.2% and 12.3% error, respectively. DeepPM also outperforms the LSTM-based model by up to 67.2% and 34.9% error for performance and power, respectively.**

***Index Terms*—Power and performance modeling, system resource prediction, transformer**

I. Introduction

For decades, diverse power/performance estimation techniques for processors have been developed for efficient system design and online management. For example, the embedded systems should utilize power/performance estimation techniques to optimize power consumption and meet the power budget requirements due to the limited battery resource. As the management efficiency highly relies on the accuracy and speed of the estimation, i.e., how to precisely predict the power consumption and execution time at runtime, earlier research has focused on many modeling techniques that abstract power/performance behaviors with related system information.

A representative method is to estimate the requirement based on either source code or compiled binaries [1]–[4]. These approaches split the target program into specific high-level software granularity and build estimation models based on profiled features of the divided elements and analyzed software flows. Because the basic block is good to analyze the software flow, many works use it as granularity and use the basic block-based model to estimate the entire program characteristics and optimize performance and energy consumption [5]–[7].

However, these techniques require prior knowledge of the divided elements, e.g., timing, power, or other relevant events. Hence, the measurement or profiling on real devices is typically needed to extract features from them by executing each program on a fine-grained simulator or real hardware. For example, to utilize many power/performance models relying on performance monitoring counter (PMC) events, users should execute the target application *at least once*, either offline or online. Due to the disadvantage that the extra processes exist, most of these approaches can be used in limited situations.

This paper proposes a framework called Deep Power Meter (DeepPM), which predicts the performance and power consumption of basic blocks in a fast and accurate manner *only using compiled binaries.* Our technique is inspired by the machine learning research in natural language processing (NLP), in particular, based on *transformer*-based deep learning techniques. The framework divides a compiled target program into basic blocks and predicts the power consumption and performance of the basic blocks by using a transformer model [8], which takes instruction sequences as the inputs. DeepPM does not need any profiling of running applications at either slow simulators or real hardware to extract features; we can quickly and accurately provide information of basic blocks, which have never been seen before, without physically running it.

We show that the transformer-based neural network structure provides highly accurate prediction even for long basic blocks with many instructions, which indeed takes the majority of actual program executions. It allows us to cover a wide range of applications, unlike the existing performance model [9] based on Long Short-Term Memory (LSTM), which only can predict short basic blocks, e.g., less than 50 instructions. In addition, our technique estimates the throughput and power consumption simultaneously using a single transformer model. We also present how to organize the transformer encoder stack for accurate prediction by considering the underlying characteristics of the input data, i.e., instructions in basic blocks. In the evaluation conducted with various SPEC2006 benchmark suites, we show that DeepPM provides a highly accurate prediction for performance and power consumption with 10.2% and 12.3% error, on average, respectively. DeepPM also outperforms the LSTM-based model [9] by up to 67.2% and 34.9% error for performance and power, respectively.

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics, Republic of Korea under Project Number SRFC-IT2002-06. The ICT at Seoul National University provided research facilities for this study. Yeseong Kim was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2018R1A5A1060031). (Corresponding authors: Yeseong Kim and Jihong Kim)

II. Background and Motivation

A. Learning-based Performance/Power Estimation

High-level/source-level software modeling has become popular for fast and accurate estimation of performance and power. These approaches construct models based on the parts of the target program divided by specific granularity. As that granularity, they perform either static program analysis of control flow graph (CFG) or dynamic modeling by performing the programs to measure target values such as power or core cycle. Because the basic block is good to analyze the software flow, many works use it as a granularity. However, building models require extracting features at least once by executing the target program on the fine-grained simulator or real hardware. This typically results in a considerable overhead to the high-level software model technique, which is infeasible in many situations that need online management and optimization.

Recent research work proposed machine learning-based techniques to profile them without executing basic blocks [9], [10]. Since their model takes the basic block itself as an input for the learning model, we can estimate the performance/energy without the execution of the basic blocks. However, these works pose several challenges in applying them to real applications.

For example, the work shown in [10] estimates the energy of the basic block using an Artificial Neural Network (ANN) model. However, it does not take into account a sequential relationship between instructions in the basic block. In addition, since the ANN only works for the inputs with a fixed number of features, it should modify the number of instructions for every basic block by inserting NULL or cutting some instructions. That is, their basic blocks are different from those used in the actual application. Also, it can not cover a long basic block with a larger number of instructions than the modeled ones.

Another research [9] proposed an LSTM-based basic block modeling technique. Since the LSTM can predict a sequence of input data, it effectively responds to the variable input size, i.e., the instruction sequence of the basic block. However, a critical shortcoming of the LSTM model is that it typically shows poor accuracy for long sequences. Due to this limitation, they showed their benefits only for short-length basic blocks.

B. Prediction quality vs. Basic block length

To better understand the length of the basic blocks in real-world applications with the relationship to the accuracy of the prior technique, we collected all the unique basic blocks of SPEC2006 benchmark suite [11]. After that, we divided basic blocks into three groups based on the number of instructions: 0% to 90%, 90% to 95%, and 95% to 99.99%, i.e., 95% to 99.99% is the group of longest basic blocks. We then ran each benchmark and counted how many times the basic blocks of each group were executed by using the pin tool [12].

Fig. 1 shows the usage portion of each group. We observe that even though the long basic blocks are a small part of the entire basic block set, they are executed frequently during the application runtime. For example, for the 470.lbm application, the basic blocks, whose number of instructions is beyond the 95% percentile, appear 58% during the application execution.

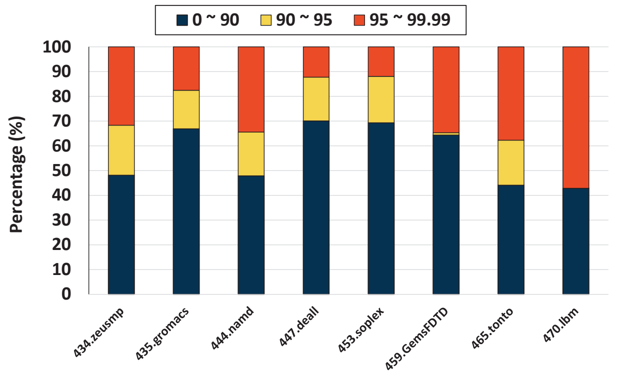


Fig. 1. The percentage of usage per basic block group when executing the benchmarks in SPEC2006. The groups are divided into three types (0% to 90%, 90% to 95%, and 95% to 99.99%) based on the number of instructions in the basic block. It shows that long basic blocks having many instructions are frequently executed.

To verify the prediction quality of the LSTM model in [9] for the long basic blocks, we train the model based on our dataset consisting of the SPEC2006's basic blocks (we describe a detailed description of our dataset in the Section IV-A). Fig. 2 summarizes the results for five groups identified by the number of instructions for each basic block. The results show that the LSTM model has a higher error for the basic blocks having more instructions. In particular, for longer basic blocks beyond 50 cycles, it exhibits unacceptable error of more than 45%.

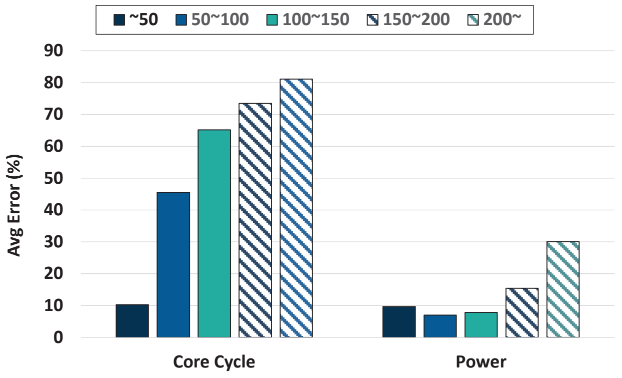


Fig. 2. Average error for each group. The groups are divided into five types (below 50, 50 to 100, 100 to 150, 150 to 200, and over 200) based on the number of instructions in the basic block. The LSTM model show higher errors for both core cycle and power as the number of instructions increases.

C. Transformer

The RNN-based models such as LSTM process input tokens sequentially and update a state vector by combining the current token with the last state vector, in which the information of previous tokens is compressed. However, some information of the input sequence could be lost in the process of compressing them into one vector. It makes the LSTM model have a lower accuracy as the sequence becomes longer.

This paper addresses the shortcomings by utilizing the transformer neural network structure, which offers much higher robustness for long sequences. The transformer [8] is developed to handle sequential input data like words or sentences mainly in NLP and computer vision (CV) fields. The transformer model solves the issues in handling the long sequences based on the attention mechanism. The attention mechanism processes all tokens at the same time by calculating attention weights between tokens in successive encoder layers. Therefore, it directly models relationships between all input tokens, such as words in a sentence, without loss of information along their length. In addition, training the transformer model is typically faster than the LSTM thanks to its high parallelism.

In our cases, the basic blocks usually have a very different length depending on the number of instructions. The instructions also have different effects depending on the distance from each other. Thus, the characteristics of the transformer, which calculates the relationship between inputs directly, enable us to consider various influences between inputs regardless of the long instruction sequences.

III. DeepPM Design

A. Overview

Fig. 3 shows the overview of the proposed DeepPM. Our framework consists of three parts at the high level: Basic Block Generator, Tokenizer, and Transformer Model. For host-compiled benchmark programs, the Basic Block Generator collects all the unique basic blocks. After that, Tokenizer proceeds tokenization for the basic blocks to generate input tokens in the form of a vector for the Transformer Model. This process is similar to the one used in NLP for the general word tokenization. Finally, using the tokenized basic blocks as inputs, we train the Transformer Model according to the target environment with measured power and core cycles. For inference, we can utilize the trained model to predict average power and core cycle only with the host-compiler target application, i.e., without running a new application on the target platform to collect additional runtime information.

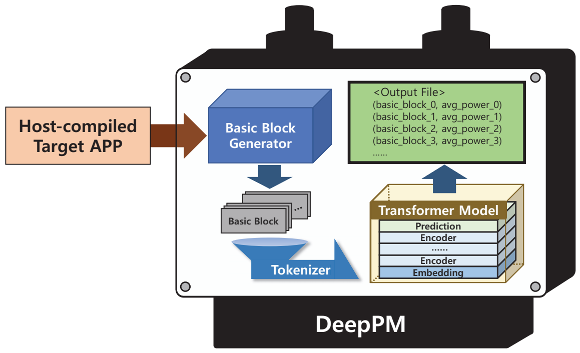


Fig. 3. Design of DeepPM Architecture.

B. Basic Block Generator & Tokenizer

The first two components, Basic Block Generator and Tokenizer, perform the process of converting the host-compiled target application, either benchmark applications or the one that the user wants to estimate through DeepPM, to the input vector of the Transformer Model. We have developed these two components in a similar method to the one used in [9].

Basic Block Generator collects all basic blocks of host-compiled target applications. For example, at Executable and Linkable Format (ELF), it extracts the executable hexadecimal machine code of each symbol included in the .text section. After that, using the dynamorio tool [13], we obtain basic blocks by dividing the code based on a control transfer instruction of any kind, whether direct, indirect, conditional, or unconditional.

Tokenizer converts the generated basic blocks into the input token vectors for the Transformer Model. We regard each of the opcodes and operands as one token. In addition to them, we also insert *source*, *destination*, *constant*, *memory*, and *end* tokens, following the generic tokenizing techniques. The *end* token is inserted each time after the last token in each instruction. If the instruction uses a source/destination operand, the *source/destination* token is inserted before that operand. If the constant/memory address has been seen, we use the *constant/memory* token to replace the value.

C. DeepPM Transformer Model Architecture

Fig. 4 shows a diagram of the DeepPM Transformer Model. The model architecture consists of the following components: embedding layer, three types of encoder layers (basic block, instruction, and opcode), and prediction layer.

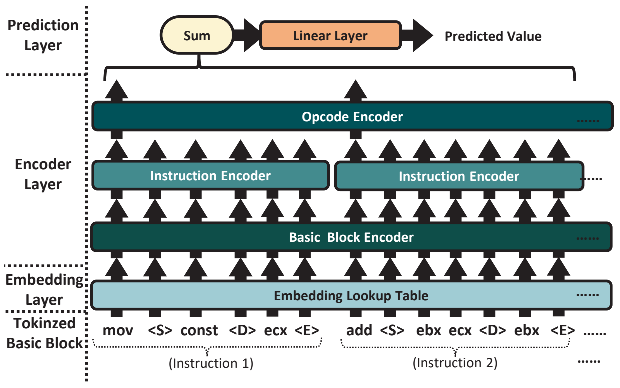


Fig. 4. DeepPM Transformer Model Architecture.

1) Embedding Layer: The embedding layer takes the sequence of tokens included in a basic block as the input. It converts each token into a d_{model} dimensional vector through an embedding table. And, the positional encoding injects the information for the order of sequence in the same way of the typical transformer. So, the output of the embedding layer is a tensor consisted of vectorized tokens of the basic block.

2) Encoder Layer: The encoder layer is composed of a stack of num_layers encoders. It receives the basic block embedding tensor from the embedding layer as an input and outputs a newly encoded tensor through encoders.

In our work, we develop a new encoder structure to consider the basic block characteristics. Fig. 5 compares the standard encoder with the DeepPM encoder. There are three major differences: (i) DeepPM uses a multi-head weighted self-attention mechanism. (ii) It also does not use layer normalization. (iii) Instead of using the same encoders in the encoder stack, DeepPM uses three different encoders according to the target relationships that each encoder aims to compute.

Multi-head weighted self-attention First, we propose the multi-head weighted self-attention mechanism, which calculates attention considering the interval between tokens. In the fields such as NLP and CV, which usually utilize the standard transformers, the distance between tokens was not a significant factor. In contrast, each token of a basic block performs different roles depending on the surrounding tokens and their locations. For example, the instructions close to each other are likely to process the same or related data. Thus, the influence it receives from other tokens is different depending on the distance. When using different weights according to the distance, we obtain better results than the standard model. The following equation shows our attention:

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

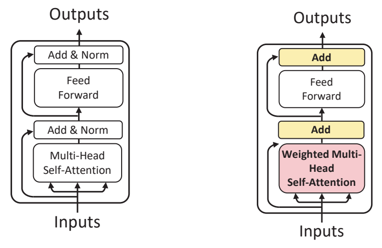
where Q, K, V are the matrix of queries, keys, values, d_k is a dimensional vector for the key, and $\circ$ is the element-wise multiplication. R is the interval weighted ratio matrix which define as:

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

where s is the length of the input sequence.

Removing layer normalization Second, DeepPM does not use layer normalization. In NLP and CV, which use the vector of words or images as input or output, the ratio between vectors is more important than the absolute values of the vector. Thus, they use layer normalization, which also offers fast learning speed by adjusting the size of the values while maintaining the ratio through normalization. However, since the goal of the DeepPM Transformer is not an output vector normalized itself, it should predict an output vector with appropriate degrees of values eventually related to the power and core cycle. In other words, to predict the target value, such as the average power or core cycle, the absolute number of the vector itself is crucial since the prediction of the final value is made through the degree of that number. Therefore, we remove layer normalization from our model and only use residual connections, as shown in Fig. 5b.

Relationship differentiation Lastly, we use three types of encoders depending on a range of calculating attention values. The standard encoder originally computes the attention between all vectors in the input tensor. That is, when the embedding tensor is the input to the encoder layer, it calculates the relationship between all tokens. However, in our case, because the basic block consists of several instructions, the relationship of the tokens included in the same instruction is more important than the one belonging to other instructions. To reflect this, we



(a) Standard Encoder (b) DeepPM Encoder

Fig. 5. Diagrams of standard encoder of typical transformer model and DeepPM encoder

designed three different encoders. One of them is a *basic block* encoder that calculates the relationship between all tokens of the basic block, and the other one is an *instruction* encoder that calculates the relationship between the tokens included in the same instruction. Also, considering that the power or core cycle of the basic block is most affected by the opcode than other factors, an *opcode encoder* was constructed for calculating the relationship between the opcode token vectors that passed through the previous two encoders. The usage of these three types of encoders can be seen in Fig. 4.

3) Prediction Layer: The prediction layer predicts a single value such as average power or core cycle by applying a linear prediction to the sum of all vectors of an input tensor. The input tensor is the output tensor of the opcode encoder, which is the last of the encoder stack in the encoder layer.

IV. Evaluation

A. Experimental Setup

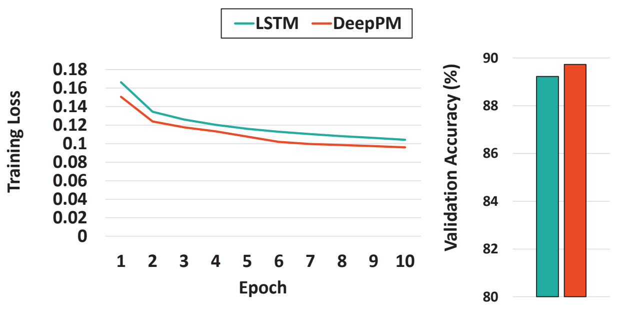
We evaluate our DeepPM framework on x86-64 Intel Skylake CPU. For benchmark applications, we collect basic blocks of 29 programs in SPEC2006 [11]. We compiled the benchmarks under the standard setting of SPEC2006 and extracted the basic blocks of the compiled binary programs through the Basic Block Generator of DeepPM explained in III-B. After that, we collected the core cycle and average power data of each basic block. To measure the core cycle of the basic block, we use the basic block core cycle profiling technique of [14]. We also implemented a power measurement tool based on Intel's Running Average Power Limit (RAPL) energy sensors. Since RAPL performs sampling of CPU energy consumption at every 1 msec [15], it does not offer sufficient granularity of measurements for a single basic block. Thus, based on the previously measured core cycle and CPU frequency, we execute multiple iterations for each basic block to achieve 1 msec, using a similar method to the one suggested in [16].

For evaluation, we used a random 80% subset of basic blocks for the training and 20% for the validation (testing). Table I summarizes the hyperparameters of the models used in the experiments. Along with the existing LSTM-based model [9] and proposed DeepPM model, we experiment four different variants of the DeepPM model to evaluate the effectiveness of the proposed transformer model structure as follows:

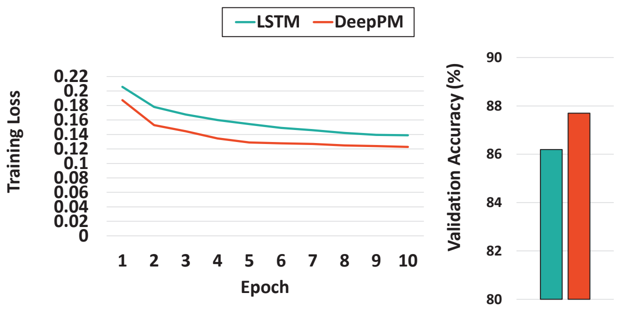
- model_A: a transformer model with standard encoders
- model_B: a transformer model with basic block encoders

TABLE I. Hyperparameters for each model.

	batch_size	dim	dim_ff	head	Encoders (num_layers)				Weighted Masking
					standard	basic block	instruction	opcode	
LSTM	4	512							
DeepPM	32	512	2048	8		2	2	4	O
model_A	32	512	2048	8	8				O
model_B						8			O
model_C						4	4		O
model_D						2	2	4	X



(a) Core Cycle Dataset



(b) Average Power Dataset

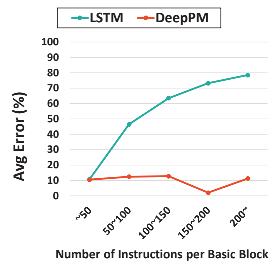
Fig. 6. Training Loss and Validation Accuracy of LSTM and DeepPM Transformer Model for core cycle and average power each.

- model_C: a transformer model with basic block encoders and instruction encoders
- model_D: a DeepPM model without the weighting in the attention

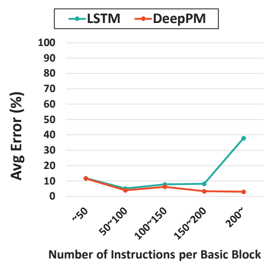
B. Accuracy Comparison

We first compare the accuracy of the DeepPM model to the LSTM-based model [9]. Fig. 6 shows the training and validation results of the DeepPM model for the extracted basic blocks. In the results, the DeepPM model provides high-quality prediction even when assuming each basic block extracted in the suites contributes equally. The error to estimate the power consumption is higher than the core cycle estimation due to the measurement noises and RAPL granularity; DeepPM achieves on average by 89.8% and 87.3% accuracy for the core cycle and power consumption, respectively. This quality of prediction is quite comparable to the state-of-art performance counter based estimation approach [17].

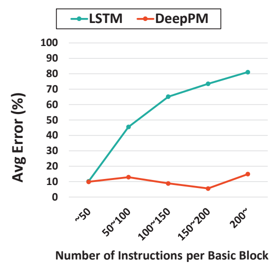
Compared to the LSTM-based model, DeepPM provides slightly better results because of the accurate prediction for long basic blocks, which frequently appear during the actual program execution as discussed in Section II-B. As discussed in Section II-B, during the actual program execution, the relatively long basic blocks appear frequently.



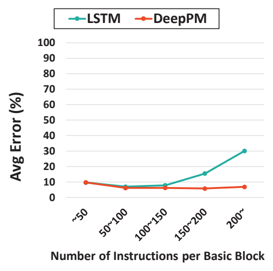
(a) Validation Dataset (Core Cycle)



(b) Validation Dataset (Power)



(c) Total Dataset (Core Cycle)



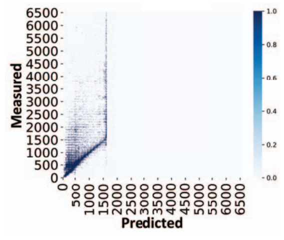
(d) Total Dataset (Power)

Fig. 7. Average Core Cycle and Power Estimation Error of LSTM Model and DeepPM Transformer Model for both Validation Dataset and Total Dataset. DeepPM shows a lower average error for groups of basic blocks with more instructions.

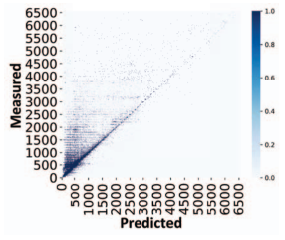
C. Prediction quality vs. Basic block length

To better illustrate the practical value of the DeepPM model, Fig. 7 summarizes the results according to the number of the instructions. The results also show that DeepPM exhibits a lower error rate than the LSTM model as the group includes more long basic blocks. For example, for the basic blocks whose number of instructions is more than 200, DeepPM significantly outperforms the LSTM-based model by 67.2% and 34.9% for the core cycle and power consumption in the validation, respectively. Thus, we conclude that DeepPM is a suitable solution to profile actual application runs on the target systems accurately.

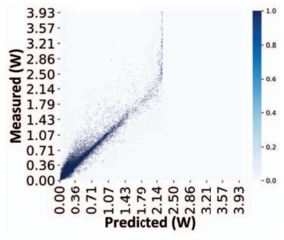
We also observed a specific region with a relatively large number of instructions that the LSTM-based model makes incorrect predictions. Fig. 8 shows the measured and predicted values for the entire dataset. As can be seen from the results



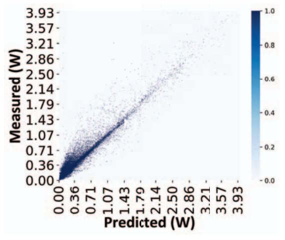
(a) LSTM, Core cycle



(b) DeepPM, Core Cycle



(c) LSTM, Power



(d) DeepPM, Power

Fig. 8. Heatmaps of measured and predicted values of LSTM and Transformer model for core cycle and power each. It can be seen that LSTM cannot predict when it exceeds a certain range, whereas DeepPM shows relatively good prediction even at large values.

of Fig. 8a and Fig. 8c, the LSTM can not predict large values beyond a certain number of instructions. However, DeepPM achieves high accuracy regardless of the number of instructions.

D. Effects of Proposed Model Learning Techniques

As discussed in Section III-C, we tuned the standard transformer/encoder structures to consider the characteristics of instructions consisting of the basic blocks. Fig. 9 is a comparative experiment result for the four different variants along with the vanilla DeepPM model. The experimental results show that each of the proposed methods contributes the performance improvements. For example, the transformer model with the standard encoders loses the training loss by 3% as compared to the proposed DeepPM model. It results in 5% lower accuracy in our evaluation for the core cycle.

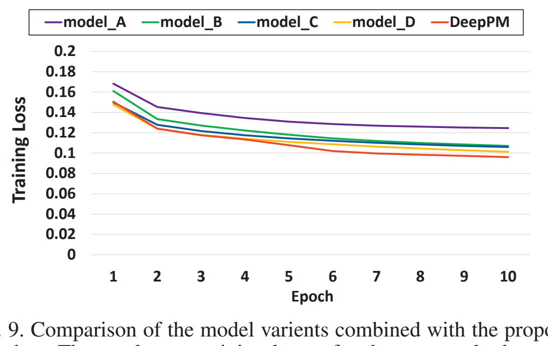


Fig. 9. Comparison of the model varients combined with the proposed encoders. The results are training losses for the core cycle dataset.

V. Conclusion

In this paper, we propose DeepPM, a performance/power prediction framework based on transformer deep learning. Unlike existing modeling work, our proposed technique does not rely on any knowledge of the runtime profiles for target applications. Inspired by the ML techniques for natural language processing, DeepPM accurately predicts the performance and power only using the compiled binary by treating the instructions as the words of the transformer model. In our evaluation, we show that the DeepPM model provides highly accurate results, 89.8% and 87.3% for the core cycle and power consumption, respectively. In addition, DeepPM also outperforms the existing LSTM-based model by 67.2% and 34.9% for the core cycle and power consumption, respectively, for long basic blocks.

References

[1] D. Lee, L. K. John, and A. Gerstlauer, "Dynamic power and performance back-annotation for fast and accurate functional hardware simulation," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 1126–1131.

[2] Z. Zhao, A. Gerstlauer, and L. K. John, "Source-level performance, energy, reliability, power and thermal (perpt) simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 2, pp. 299–312, 2016.

[3] X. Zheng, L. K. John, and A. Gerstlauer, "Lacross: Learning-based analytical cross-platform performance and power prediction," *International Journal of Parallel Programming*, vol. 45, no. 6, pp. 1488–1514, 2017.

[4] S. Shamas, M. A. Pasha, and S. Masud, "A hybrid instruction and functional level energy estimation framework for embedded processors," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[5] D. Shin, J. Kim, and S. Lee, "Intra-task voltage scheduling for low-energy hard real-time applications," *IEEE Design & Test of Computers*, vol. 18, no. 2, pp. 20–30, 2001.

[6] Z. Wang and J. Henkel, "Accurate source-level simulation of embedded software with respect to compiler optimizations," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 382–387.

[7] F. Alam, P. R. Panda, N. Tripathi, N. Sharma, and S. Narayan, "Energy optimization in android applications through wakelock placement," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–4.

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

[9] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on machine learning*. PMLR, 2019, pp. 4505–4515.

[10] T. Hönig, B. Herzog, and W. Schröder-Preikschat, "Energy-demand estimation of embedded devices using deep artificial neural networks," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 617–624.

[11] "Standard Performance Evaluation Corporation. Spec CPU benchmark suite, 2006," https://www.spec.org/cpu2006/.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[13] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization, 2003.* IEEE, 2003, pp. 265–275.

[14] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sỳkora, S. Amarasinghe, and M. Carbin, "Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 167–177.

[15] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 3, 2016.

[16] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[17] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11, p. 12.