Towards On-The-Fly Code Performance Profiling

XING HU, School of Software Technology, Zhejiang University, China
WEIXIN LIN, College of Computer Science and Technology, Zhejiang University, China
ZHUANG LIU, School of Software Technology, Zhejiang University, China
XIN XIA*, Zhejiang University, China
MICHAEL LING, Huawei Technologies Canada, Canada
YUAN WANG, Huawei Sweden Research Center, Sweden
DAVID LO, School of Computing and Information Systems, Singapore Management University, Singapore

Improving the performance of software applications is one of the most important tasks in software evolution and maintenance. In the Intel Microarchitecture, CPUs employ pipelining to utilize resources as effectively as possible. Some types of software patterns or algorithms can have implications on the underlying CPU pipelines and result in inefficiencies. Therefore, analyzing how well the CPU's pipeline(s) are being utilized while running an application is important in software performance analysis. Existing techniques, such as Intel VTune Profiler, usually detect software performance issues from CPU pipeline metrics after the software enters production and during the running time. These techniques require developers to manually analyze monitoring data and perform additional test runs to obtain relevant information about performance problems. It costs a lot of time and human effort for developers to build, deploy, test, execute, and monitor the software.

To alleviate these problems, we propose a novel approach named PGProf to predict the CPU pipeline before execution and provide the profiling feedback during the development process. PGProf exploits the graph neural networks to learn semantic and structural representations for C functions and then predict the fraction of pipeline slots in each category for them during the development process. Given a code snippet, we fuse different types of code structures, e.g., Abstract Syntax Tree (AST), Data Flow Graph (DFG), and Control Flow Graph (CFG) into one program graph. During offline learning, we first leverage the gated graph neural network to capture representations of C functions. PGProf then automatically estimates the final pipeline values according to the learned semantic and structural features. For online prediction, we predict pipeline metrics with four category values by leveraging the offline trained model. We build our dataset from C projects in GitHub and use Intel VTune profiler to get profiling information by running them. Extensive experimental results show the promising performance of our model. We achieved absolute result of 49.90% and 79.44% in terms of *Acc*@5% and *Acc*@10% with improvements of 8.0%-42.7% and 7.8%-20.1% over a set of baselines.

Additional Key Words and Phrases: Program Graphs, Performance Profiling, CPU utilization, Graph Neural Networks

*Corresponding Author: Xin Xia.

Authors' addresses: Xing Hu, xinghu@zju.edu.cn, School of Software Technology, Zhejiang University, Ningbo, Zhejiang, China; Weixin Lin, martinlwx@zju.edu.cn, College of Computer Science and Technology, Zhejiang University, HangZhou, Zhejiang, China; Zhuang Liu, liuzhuang@zju.edu.cn, School of Software Technology, Zhejiang University, Ningbo, Zhejiang, China; Xin Xia, xin.xia@acm.org, Zhejiang University, Hangzhou, Zhejiang, China; Michael Ling, michael.ling@huawei.com, Huawei Technologies Canada, Canada; Yuan Wang, yuan.wang1@huawei.com, Huawei Sweden Research Center, Sweden; David Lo, davidlo@smu.edu.sg, School of Computing and Information Systems, Singapore Management University, Singapore.

ACM 1557-7392/2025/3-ART
https://doi.org/10.1145/3725212

1 INTRODUCTION

During software evolution and maintenance, it is important to improve the performance of software applications. Developers spend a lot of time on performance management to identify performance bottlenecks (or hot spots), such as hardware issues, threading issues, and I/O traffic. Unfortunately, identifying performance limitations from hardware usage like CPU pipeline usage has remained an open problem in the performance engineering field [51]. The CPU pipeline is a series of sequential steps from the instructions that attempts to keep every part of the processor busy. Understanding how efficiently the code is passing through the core pipeline is critical for developers to identify hardware-level performance problems in the software [1]. For example, linked data structures are commonly used in the software, but cause idleness in the pipeline while data is retrieved and there are no other instructions to execute, thus result in utilization inefficiencies of the CPU pipeline. Generally, the available CPU execution pipeline is split into four basic categories: *Retiring*, *Bad Speculation*, *Front-end Bound* and *Back-end Bound* based on hardware operations (*μOps*) flow through a pipeline slot [1, 51]. Analyzing metrics for these four categories helps developers to focus on the top hotspots of the application.

Profiling is one of the most popular techniques to collect such performance information [17, 27]. Generally, profilers detect software performance behaviors described above after software enters production [10, 11, 20]. For example, VTune Profiler [1] collects a complete list of events for analyzing a typical client application. It calculates a set of predefined ratios used for these four metrics and facilitates identifying hardware-level performance problems, such as the Front-End stall and its causes.

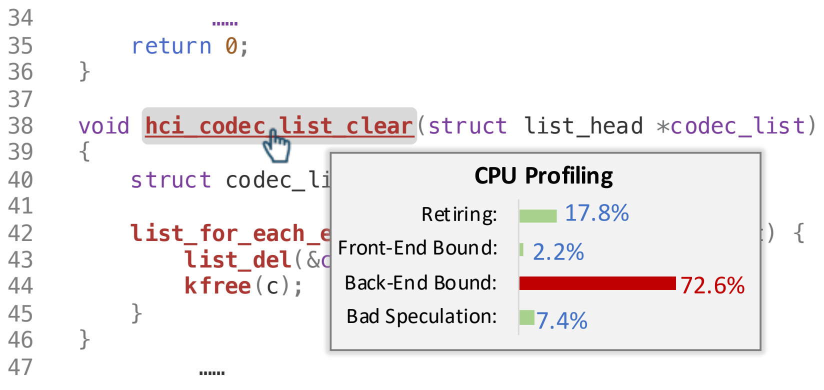
However, developers spend lots of time and effort to identify performance issues by exploiting profiling techniques. To analyze the performance of the source code, they should collect the profiling information during the runtime. This process is time-consuming, especially for a large system. First, running a large software system requires a lot of time on deployment and build. Second, locating the performance metrics for each function is time-consuming and labor-consuming. Developers have to construct enough test cases to cover every function, thereby ensuring each function can be run. Third, if a function is detected as a bottleneck, developers usually need to optimize it. Then, the system has to be re-run and tested to get new performance metrics. Therefore, it is thus highly desirable to have a tool that provides automatic performance profiling before the software enters production and provides performance feedback during the development process (e.g., shows pipeline metrics in IDEs for developers as shown in Figure 1). The on-the-fly performance profiling tool can improve performance optimization efficiency significantly.

In this paper, we refer to the task that performs the performance profiling during development as "On-the-Fly Performance Profiling" and propose a novel approach named PGProf to automate this task. It can predict four metric values of the CPU pipeline given a C function in real-time during development. These values are four percentages that add up to 100%. PGProf can help developers discover performance issues and optimize the source code in real time. We illustrate a usage scenario of PGProf as follows:

Without Our Tool: Consider a developer Alice. Daily, when Alice writes a C program to complete a coding task, she wants to know the CPU profiling information of the written functions. To get this, Alice should first write test cases for these functions and execute the whole software. Then, she uses a profiler to collect the profiling values. Furthermore, if this function has a bottleneck on a specific aspect, Alice has to optimize her code and repeat the above process until the code has no bottleneck on the CPU. It costs a lot of time and human efforts through the traditional profiling process.

With Our Tool: Now consider Alice adopts our PGProf and she is working on the IDE. Just like Figure 1 shows, Alice selects the function and chooses to profile it. The IDE will ask our model PGProf to get the CPU profiling values (i.e., *Retiring*: 17.8%, *Front-End Bound*: 2.2%, *Back-End Bound*: 72.6%, and *Bad Speculation*: 7.4%)

[1]https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html



```
        ......
    return 0;
}

void hci_codec_list_clear(struct list_head *codec_list)
{
    struct codec_li

    list_for_each_e                                    ) {
        list_del(&c
        kfree(c);
    }
}
        ......
```

CPU Profiling
Retiring: 17.8%
Front-End Bound: 2.2%
Back-End Bound: 72.6%
Bad Speculation: 7.4%

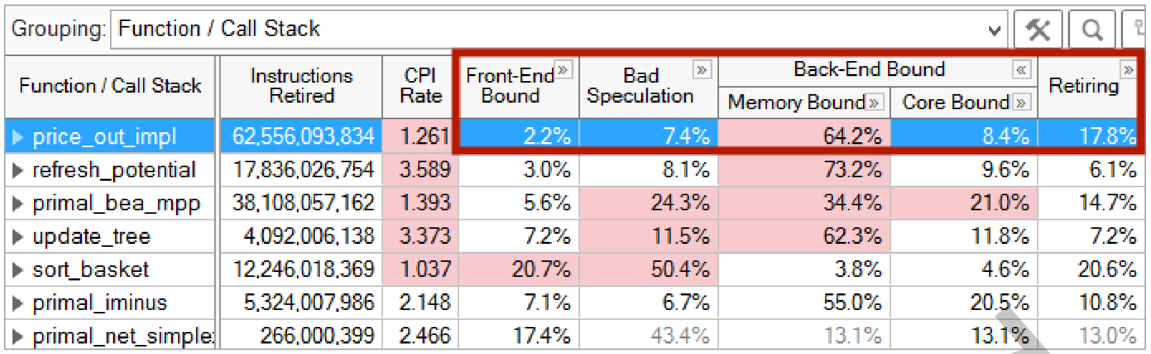
Fig. 1. The usage scenario of our approach PGProf

without writing test cases and execution. With the help of our tool, Alice successfully detects the bottleneck (i.e., Back-End Bound: 72.6%) of the function and optimizes her code on the fly.

According to Cito et al. [11] and Ahmed et al. [2], collected information by monitoring tools can be utilized to build performance models and identify performance issues. Inspired by these studies, we propose a neural network to learn from the collected performance information and use it to predict pipeline metrics. Although prior studies [35, 39] proposed to utilize deep learning techniques to deal with software performance issues (e.g., compiler optimization), there is almost no prior work at predicting CPU profiling metrics.

Intuitively, the performance of the source code is related to the code structures, such as syntax trees, control flows, and data flows. Learning from these structures simultaneously is quite challenging. To learn these code structures, we propose to represent the source code as a graph. Similar to Allamanis et al [5], we build the program graph by adding edges to the ASTs. However, they only add edges according to variables uses and updates and they do not use control flow information well. Different from them, we construct program graphs by fusing different code structures, e.g., Abstract Syntactic Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG). To predict the CPU metrics, the model needs to generate four values that add up to 100%. Existing works usually exploit neural networks to solve classification task (e.g., clone detection [29, 46]) or generation tasks (e.g., code completion [31]). The prediction target of our study is different from theirs. Thus, the prediction target of neural networks should be adapted to our task. To effectively predict the four metric values, we formulate it as a distribution estimation task.

PGProf consists of three phases: program graph construction, model training, and online application. During the program graph construction phase, we build program graphs by fusing AST, CFG, and DFG of the given program. Then, we train the graph neural model on these program graphs of C functions extracted from 62 C projects mined from the GitHub. We build these projects and run them by executing the generated test cases. Next, we leverage the profiling tool named Intel VTune to get the pipeline metrics for these C functions. When it comes to online prediction, for a given C function, we fit them into the trained PGProf model to estimate its performance metrics. To verify the suitability of our proposed model, we conduct extensive experiments on the dataset. We achieved absolute result of 49.90% and 79.44% in terms of *Acc@5%* and *Acc@10%*. By comparing with several baselines, the superiority of our proposed PGProf model is demonstrated. In summary, this work makes the following main contributions:


Grouping: Function / Call Stack

Function / Call Stack	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound		Retiring
					Memory Bound	Core Bound	
price_out_impl	62,556,093,834	1.261	2.2%	7.4%	64.2%	8.4%	17.8%
refresh_potential	17,836,026,754	3.589	3.0%	8.1%	73.2%	9.6%	6.1%
primal_bea_mpp	38,108,057,162	1.393	5.6%	24.3%	34.4%	21.0%	14.7%
update_tree	4,092,006,138	3.373	7.2%	11.5%	62.3%	11.8%	7.2%
sort_basket	12,246,018,369	1.037	20.7%	50.4%	3.8%	4.6%	20.6%
primal_iminus	5,324,007,986	2.148	7.1%	6.7%	55.0%	20.5%	10.8%
primal_net_simple	266,000,399	2.466	17.4%	43.4%	13.1%	13.1%	13.0%

Fig. 2. An example of percentage of pipeline slots in each category for the whole application.

- We propose a novel graph-based model, PGProf, to automatically predict CPU pipeline metrics given a C function before execution. It can help developers to identify hot spots in source code during the development process.
- We build program graphs by combining multi-types of code structures that can help to learn code representations effectively.
- We build a dataset with C functions and their corresponding fractions of pipeline metrics for four categories from real-world projects. To the best of our knowledge, this is the first dataset for this task.
- We released our replication package [2].

2 BACKGROUND

In this section, we briefly introduce the CPU pipeline metrics, task definition, and graph neural network models.

2.1 CPU Pipeline Metrics

Generally, profilers use on-chip Performance Monitoring Units (PMUs) to obtain how an application is utilizing available hardware resources and how to make it take advantage of CPU microarchitectures. Modern CPUs employ pipelining as well as techniques like hardware threading, out-of-order execution and instruction-level parallelism to utilize resources as effectively as possible. Each pipeline slot available during an application's runtime will be classified into one of four categories [51]:
Retiring. Retiring metric represents a Pipeline Slots fraction utilized by useful work, meaning the issued micro-ops (μOps) that eventually get retired.
Back-End Bound. Back-End Bound metric represents a Pipeline Slots fraction where no μOps are being delivered due to a lack of required resources for accepting new μOps in the Back-End.
Front-End Bound. Front-End Bound metric represents a slots fraction where the processor's Front-End undersupplies its Back-End.
Bad Speculation. Bad Speculation metric represents a Pipeline Slots fraction wasted due to incorrect speculations.

Each metric is an event ratio defined by Intel architects and has its own predefined threshold. Intel VTune Profiler is a tool that analyzes a ratio value for each aggregated program unit (for example, a C function). Figure 2 shows an example of pipeline slots collected by VTune Profiler [3]. We observe that these metrics add up to

[2]https://figshare.com/s/e5470f8e454f8dd2d825
[3]https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html

100%. Through the metrics, we can find the potential problems and bottlenecks (e.g., the highlighted metric value Memory Bound in *Back-End Bound* for `price_out_impl` indicates the potential problem areas). Notice that three of the four categories, *Bad Speculation*, *Front-End Bound*, and *Back-End Bound*, all sound fairly ominous. Each of these categories represent stalls within the CPU pipeline, that is, unproductive cycles. Therefore, the percentage attributed to these categories are a less-is-better metric. On the other hand, the fourth category, *Retiring*, represents forward progress in the CPU pipeline, that is productive cycles. Therefore, the percentage attributed to the *Retiring* category is a higher-is-better metric [1].

Predicting software performance especially execution time at the source code level has been proposed in recent years [10, 23]. The long execution time represents the "result" of the existence of performance issues in the programs, but it cannot help developers to understand these issues. The CPU metrics are used as low-level supplementary information to assist developers to identify the issues causing long execution time. For example, the Back-End Bound issue occurs when a program (See Listing 1) makes a large number of memory accesses and will spend significant time waiting for them to finish. Because it results in column-major order traversal of the matrix, which does not exploit spatial locality and hurts the program's performance. It means that to further improve its performance, we likely need to improve how we access memory, and reduce the number of such accesses.

Listing 1 Unfriendly memory accesses (has obnormal Back-End Bound value).

```
for(column=0; column<NUMCOLUMNS; column++)
  for(row=0; row<NUMROWS; row++)
    matrix[row][column]=row+column;
```

According to the Back-End Bound metric, we can improve our program as Listing 2 since it accesses the elements of the matrix in the order in which they are laid out in memory (row-major traversal).

Listing 2 Friendly memory accesses.

```
for(row=0; row<NUMROWS; row++)
  for(column=0; column<NUMCOLUMNS; column++)
    matrix[row][column]=row+column;
```

2.2 Task Definition

The motivation of our work is to automatically predict CPU profiling values for function-level C programs during the development environment. These profiling values (including *Retiring*, *Back-End Bound*, *Front-End Bound*, and *Bad Speculation*) add up to 100%. We need to predict these four profiling values. Generally, neural networks can effectively solve classification tasks (e.g., clone detection task [29]) and generation tasks (e.g., program repair task [53]) in the software engineering field. Different from them, we formulate this task as a probability distribution learning as per below. Let **C** be the C function and $\mathbf{V} = [v_r, v_{fb}, v_{bb}, v_{bs}]$ in which $v_r + v_{fb} + v_{bb} + v_{bs} = 1$ be the pipeline metrics. Our target is to automatically predict values of **V**. The output is normalized between 0 and 1 (instead of between 0% and 100%) as we formulate the four values prediction as the probability distribution estimation task. In other words, our goal is to train a model θ using **C** such that the distribution $\mathcal{D}'$ of predicted CPU profiling metrics $\mathbf{V}'$ is consistent with the distribution $\mathcal{D}$ of **V** as much as possible. Mathematically, our task is defined as finding $\overline{y}$, such that:

$$\overline{y} = \text{argmin}_\theta |\mathcal{D}' - \mathcal{D}| \tag{1}$$

where $|\mathcal{D}' - \mathcal{D}|$ represents the difference between distributions. In this paper, we use $AbsoluteError$ (See section 3.2.5) to measure the distance between two distributions.

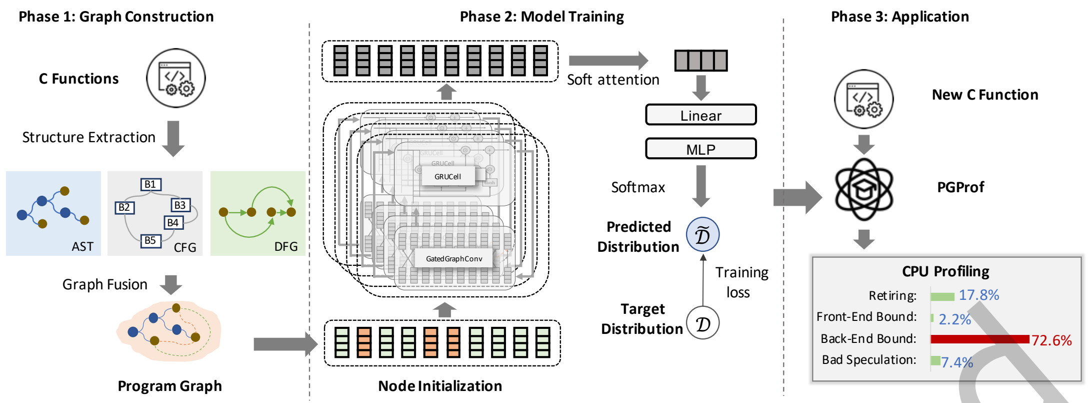


Fig. 3. The overall architecture of PGProf

2.3 Gated Graph Neural Network

The Gated Graph Neural Network (GGNN) [30] is one of the Message Passing Neural Networks (MPNNs) [15]. It mainly contains two phases: a message passing phase and a readout phase. In the message passing phase, the model propagates the messages through the edges in the graph. For each node in the graph, it aggregates messages from all its neighbors and updates its own hidden state (also called node vector or node embedding). The message passing phase will run multiple timesteps. In the readout phase, it will take all the nodes' hidden states and compute a feature vector as the graph-level representation.

Modern software is so complicated that it is difficult to learn the patterns behind the code fragment and infer the profiling metrics just at the source code level. To learn the syntax and semantic information of function-level C programs, we represent each program as a directed graph with multiple edge types. To learn the program graph well, we exploit the GGNN on this task.

3 APPROACH

In this section, we present the details of our proposed model. We first introduce the overall architecture for PGProf. Then, we describe the details of each phase of our approach. The overall framework of our approach is illustrated in Figure 3. We predict the CPU profiling according to the source code. Our approach PGProf consists of three phases, including Graph Construction, Model Training, and Application. Recently, neural networks have been widely used to capture the code features by encoding code into vectors [18, 52]. We first process the source code into different types of structures of the source code, including AST, DFG, and CFG and then fuse them into one program graph. The motivation comes from the implication that the AST can capture the variables flow information well [36]. However, some similar functions may have nearly identical AST structure but with varying profiling metrics, such as the loop interchange optimization on the matrix-multiply example discussed in [51]. In the Model Training Phase, we exploit the GGNN to encode the program graph to get the representation of the C function and predict the profiling metrics by estimating the metric distribution. In the Application Phase, the trained PGProf takes the new C function as input and predict its metrics.

3.1 Phase 1: Program Graph Construction

To construct the program graph $\mathcal{G}$, we first build Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG) from the C function. In this study, we exploit the Tree-sitter[4] to parse the source code and then extract them. Similar to Allamanis et al. [5], we fuse them by adding different types of edges (i.e., `NCS` edges and `ComputeFrom`) to the AST. To learn a much richer code structure, we extend the program graph in: 1) We use the `NextUse` edge to represent the variable usages instead of the `LastWrite` and `LastRead` edges to simplify the program graph; 2) We include the CFG by adding the `MustExe` edges and `MayExe` edges. The details of our program graph construction process are described as follows.

The AST consists two types of nodes, non-terminal nodes (represent the program's grammar structures) and terminal nodes (represent code tokens). All nodes $\mathcal{V}$ in the program graph $\mathcal{G}$ come from the AST nodes. As shown in Figure 4, the black edges represent the AST relationships between nodes. Considering the semantic information in the source code, we add natural code sequence (`NCS`) edges (the green dotted edges in Figure 4) by connecting the terminal nodes in the AST. These connected nodes are composed into a sequence of the source code tokens.

Then, we add data flow edges to the graph that illustrates how variables flow through the program. Specifically, we extract two types of data flows from the DFG, i.e., `NextUse` edges (orange dotted edges) and `ComputeFrom` edges (purple dotted edges). The `NextUse` edges demonstrate how the same one variable flows in the program. The `ComputeFrom` edges demonstrate the computation information between different variables, e.g., `a⇢b` in the Figure 4 means that variable `b` is computed from variable `a` in the `int b = a * 2;` statement.

Except for DFG, we also integrate the CFG into the program graph. The CFG illustrates all paths that might be traversed through a program during its execution. The edges in CFG we used in this paper include two types, `MustExe` (blue dotted edges) and `MayExe` (red dotted edges). `MustExe` edges represent that the paths will definitely be executed and `MayExe` edges represent the paths will be executed under specific conditions.

Finally, we get a program graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, in which nodes $\mathcal{V}$ come from the AST nodes and edges $\mathcal{E} = (\mathcal{E}_{AST}, \mathcal{E}_{NCS}, \mathcal{E}_{NextUse}, \mathcal{E}_{ComputFrom}, \mathcal{E}_{MustExe}, \mathcal{E}_{MayExe})$.

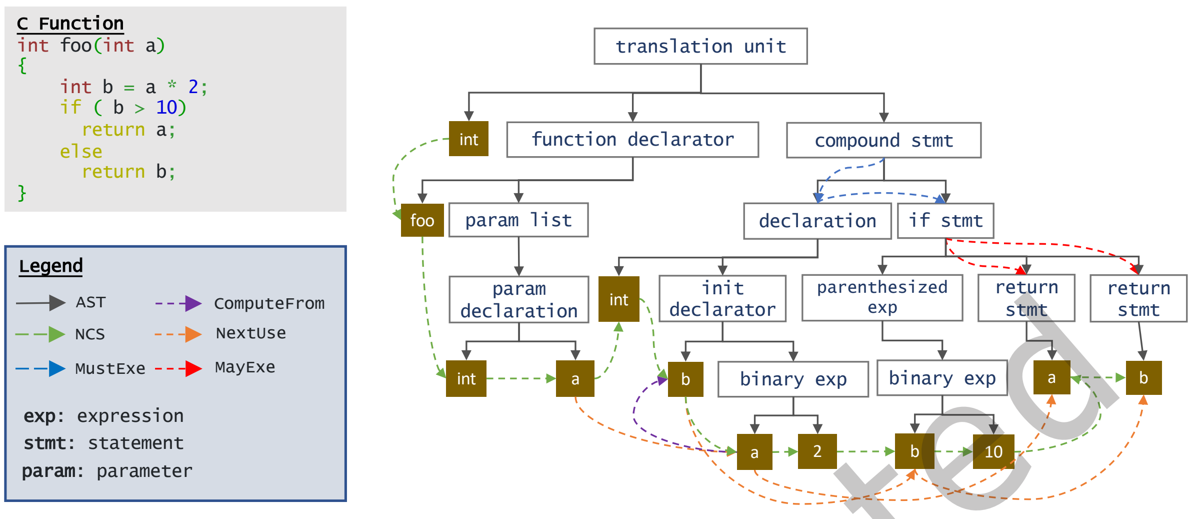
3.2 Phase 2: Model Training

In the model training phase, we exploit the graph neural network to learn code representations from the graph built in Phase 1. The model includes five modules, i.e., Node encoder, Message passing, Graph-level readout pooling layer, Multi layer perceptron, and the loss Function. The details of these modules are shown as follows:

3.2.1 *Node encoder.*

The node encoder module aims to convert graph nodes into embeddings. The nodes in the program graph $\mathcal{G}$ come from the AST and include non-terminal nodes and terminal nodes. According to the C grammar, the non-terminal nodes only have type information and terminal nodes have both type information and value information (i.e., the code tokens). In total, there are 144 node types in C programs. As shown in Figure 3, we exploit different strategies to encode different graph nodes. Before encoding them, we first build the vocabulary for them. We leverage the RoBERTa [32] tokenizer provided by UniXCoder [19] to tokenize the values of terminal nodes. The RoBERTa tokenizer splits tokens into multiple sub-tokens. After tokenization, we get 51,416 tokens in the vocabulary. Then, we extend the vocabulary by adding 144 node types and the extended vocabulary size is 51,560. Finally, we map each token to a vector representation, constructing an embedding layer, and the weights are initialized with UniXCoder's token embedding layer except the extended part's weights. For non-terminal nodes, we encode them into embeddings based on the corresponding node types. For each terminal node, we start by tokenizing its value, which may result in multiple sub-tokens. In this case, we aggregate the embeddings of each sub-token by adding them element-wise.

[4]https://tree-sitter.github.io/tree-sitter/.


C Function

```
int foo(int a)
{
    int b = a * 2;
    if ( b > 10)
      return a;
    else
      return b;
}
```

Fig. 4. The procedure of building a graph given a program.

Finally, given a graph with $\mathcal{V}$ nodes, we can get the initial node embedding matrix:

$$\mathbf{H}^{(0,0)} = [\mathbf{h}_1^{(0,0)}, \mathbf{h}_2^{(0,0)}, ..., \mathbf{h}_{\mathcal{V}}^{(0,0)}] \tag{2}$$

where $h_v^{(0,0)}$ is the initial node embedding of node v in the first step of the first GGNN layer.

3.2.2 Message passing. In this study, we exploit the GGNN to learn from our program graph that is a directed graph with different edge types. Existing studies have shown that GGNN can effectively learn from such graphs [5, 43]. To better capture the program graph information, we implement our PGProf by stacking multiple GGNNs. The model architecture of our proposed PGProf is illustrated on the Figure 3.

In each layer, the GGNN unrolls the Recurrent Neural Network (RNN) for $\mathcal{T}$ steps and does back-propagation. At each layer k, a node v get messages from its neighbor nodes (denoted as $\mathcal{N}(v)$ below) at each time step t. These messages are aggregated by using linear transformation on the neighbor node embeddings:

$$a_v^{(k,t)} = \sum_{u \in \mathcal{N}} W_{e_{vu}}^{(k)} \mathbf{h}_u^{(k,t)} \tag{3}$$

where a_v is the final aggregated result for node v, e_{vu} are all the incoming edges with a specific edge type, and $W_{e_{uv}}$ is the learned matrix that is the same for a specific edge type.

After aggregating these messages, the node v needs to incorporate two different types of information (i.e. the aggregated result from the neighbors $a_v^{(k,t)}$ and its own node hidden state in the last timestep $\mathbf{h_v}^{(k,t)}$). The hidden state of node v is updated as: $\mathbf{h}_v^{(k,t+1)} = GRU(a_v^{(k,t)}, \mathbf{h_v}^{(k,t)})$.

The same as the GGNN, we also use Gated Recurrent Unit (GRU) [9] to update the hidden states. By aggregating messages and updating, the node v can learn the local graph structural well. All nodes in the program graph will be updated in the same way. The updated node hidden states in the k layer is passed as input to the next $k + 1$ layer.

3.2.3 Graph-level readout pooling layer. After running the message passing phase for $\mathcal{T}$ steps, we should get the representation of the graph layer. Considering that different nodes contribute differently to the graph, we use the soft-attention mechanism to get the graph representation:

$$\mathbf{H}_{\mathcal{G}} = \sum_{v=1}^{\mathcal{V}} softmax(f_{gate}(\mathbf{h}_v^{(\mathcal{T})}))\mathbf{h}_v^{(\mathcal{T})} \tag{4}$$

where $\mathbf{H}_{\mathcal{G}}$ is the graph representation and f_{gate} is used to compute attentions score for each node. In this paper, we retrieve the last GGNN layer's node embeddings in our model and apply the readout function on all nodes to get a graph representation (i.e., $\mathbf{H}_{\mathcal{G}}$).

3.2.4 Multi Layer Perceptron. To further capture latent features in the function representation, we add a Multi-Layer Perceptron (MLP) on the function embedding vector $\mathbf{H}_{\mathcal{G}}$. It takes $\mathbf{H}_{\mathcal{G}}$ as input and outputs a vector $\mathbf{z}$ with the same dimension as $\mathbf{H}_{\mathcal{G}}$: $\mathbf{z} = a(W_1\mathbf{H}_{\mathcal{G}} + b_1)$ where a is ReLU in this paper.

Finally, we add a linear layer and a Softmax layer to map $\mathbf{z}$ into a vector with a dimension of four to represent the values of the four CPU pipeline categories:

$$\widetilde{\mathbf{V}} = [\widetilde{v}_r, \widetilde{v}_{fb}, \widetilde{v}_{bb}, \widetilde{v}_{bs}] = Softmax(W_2\mathbf{z} + b_2) \tag{5}$$

3.2.5 Loss Function. In this paper, we adopt the least absolute deviations (L1) loss to estimate the error between the predicted values $\widetilde{\mathbf{V}}$ and the real values $\mathbf{V}$. L1 Loss function minimizes the absolute differences between the estimated values and the existing target values:

$$S = \sum_{i=1}^{4} |\widetilde{v_i} - v_i| \tag{6}$$

where $\widetilde{v_i}$ and v_i indicate the ith value in $\widetilde{\mathbf{V}}$ and $\mathbf{V}$, respectively.

3.2.6 Model training. After building the PGProf model, we train it by minimizing the loss function. To improve the robustness of our model, we apply the dropout [40] technique during training.

3.3 Application

After the training process, we can get a trained neural network PGProf to predict CPU pipeline metrics for new C functions during the development. As Figure 1 shows, trained PGProf can be integrated into the IDE and provide performance feedback during the development process. When developers choose to profile a C function during the development process, the PGProf can provide CPU profiling data for them in the development environment.

4 DATASET PREPARATION

In this paper, we consider predicting CPU pipeline metrics for C functions. Figure 5 illustrates the dataset construction process that contains five steps. To train a deep learning based profiling prediction model, we need to construct a dataset with $\langle function, profiling\rangle$ pairs in which $profiling$ has four values for four categories. We collected open-source C projects from GitHub, built them, executed provided test cases on them, and got profiling information by using VTune. The details of dataset construction are shown as follows:

Step 1. Mining C Projects. In this paper, we build a real-world dataset that is collected from GitHub. To ensure the quality of C projects, we collected popular C projects according to the number of their stars. Specifically, we sort all C projects in GitHub by their stars in descending order. Projects with fewer than 100 stars are not be considered. The language statistics may be incorrect due to library usage and code changes [5] which may introduce bias, for example, projects may gain stars for reasons unrelated to their codebase quality; thus, we also reviewed some C-related projects from well-known companies such as Microsoft, Google, and Facebook.

[5]https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-repository-languages

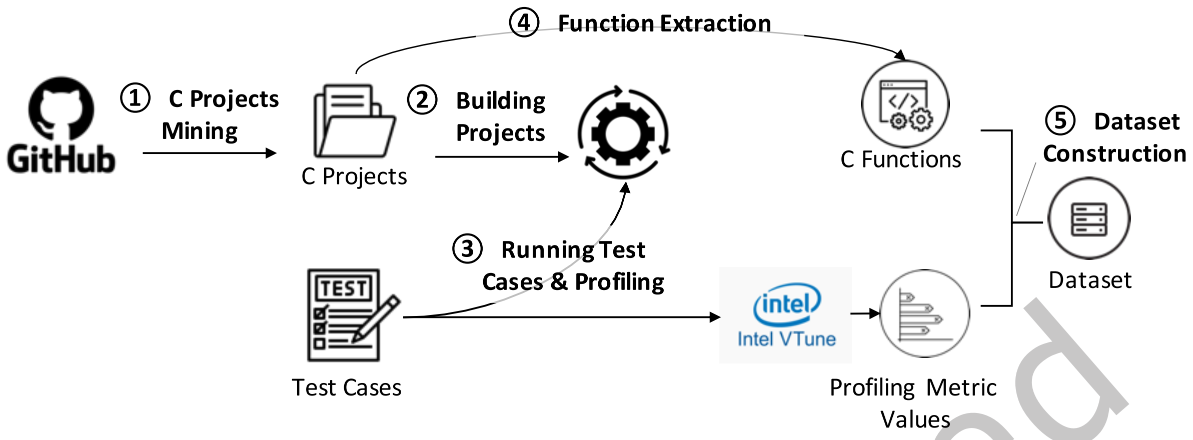


Fig. 5. The dataset construction process

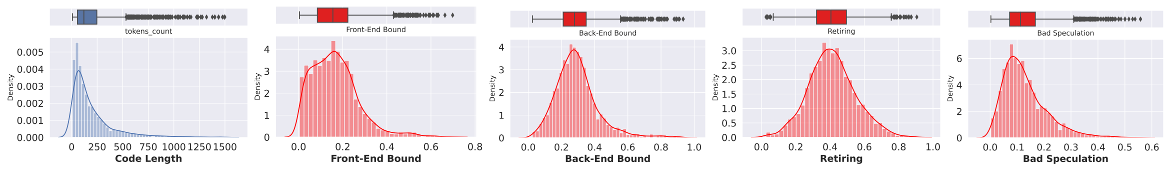
Initially, a total of 240 C projects were considered. The distribution of stars is shown in Table 1. However, it is necessary to exclude some C projects without configuration documentation and test cases as we need to build these projects and run test cases to get the profiling information. Consequently, 134 out of the initial 240 C projects are removed.

Table 1. The dataset's star statistics

Metric	Stars
Mean	4,321
Min	100
25%	850
50%	1,644
75%	3,730
Max	66,928

Step 2. Building C Projects. We compiled and built each project manually according to the building documentation. We used four tools to build them, including Makefiles, CMake[6], Ninja[7], and Meson[8] according to the building descriptions from these projects. To mitigate the effects of compiler's optimization, we use -O0 optimization flag in each project. In addition to setting the same optimization level, we also try our best to use the same compiler options. However, many projects failed to build since ① Lack of detailed building documentation; ② Missing dependencies; ③ Failed when regenerating configure files with Autoconf. In this stage of the data collection process, 20 C projects that fail to build are removed. Additionally, in the interest of promoting consistency in our methodology, we also excluded three C projects that necessitated the use of distinct versions of build tools. Finally, we successfully built 83 projects.

[6]https://cmake.org/.
[7]https://ninja-build.org/.
[8]https://mesonbuild.com/.



(a) Code length distribution (b) Front-End Bound distribution (c) Back-End Bound distribution (d) Retiring distribution (e) Bad Speculation distribution

Fig. 6. Length distribution and pipeline metrics distributions of the training data

Step 3. Running Test Cases and Profiling. This step is responsible for collecting CPU pipeline metrics of C functions by running test cases. For each project, we manually inspect its content and use all the test cases available. In this paper, we exploit the Intel VTune Profiler to monitor the CPU pipeline performance metrics for C functions. As the VTune Profiler monitored the performance by sampling during the collection duration, for each run, we continuously executed test cases until stable performance samples are collected. Each sample interval lasts at least 10s for each test case. Each project was run ten times and the average standard deviation of ten groups of profiling values is 0.036 after removing outliers. The individual standard deviation for each metric is: *Front-End Bound* - 0.037, *Back-End Bound* - 0.060, *Retiring* - 0.042 and *Bad Speculation* - 0.030.

The outliers are detected by the Modified Z-Scores [25] that can measure outlier strength or how much a particular score differs from the typical score. Similar to the standard Z-Scores, both of them can be used to detect outliers. However, the Modified Z-Scores is more robust, especially when the number of samples is small [25]. To calculate the Modified Z-Scores, we need to compute the Median Absolute Deviation(MAD) first:

$$\text{MAD} = \text{median}(|x_i - \text{median}(\mathbf{x})|) \tag{7}$$

Where median means getting the median, x_i refers to the i-th sample's value, and $\mathbf{x}$ refers to all the samples. Then, we can calculate the Modified Z-Scores [25]:

$$M_i = \frac{0.6745(x_i - \text{median}(\mathbf{x}))}{\text{MAD}} \tag{8}$$

Where M_i indicates the i-th sample's score. We follow Iglewicz and Hoaglin [25] and regard samples with $|M_i| > 3.5$ as the outliers.

Then, we group the collected profiling data by function and use the Modified Z-Scores to detect and remove outliers within each group. A profiling data is regarded as valid if it has no outliers on all the categories of pipeline metrics. This filtering process only removes profiling data within each group, and will not reduce the number of functions. The percentage of profiling data got discarded is roughly 5%.

We use the average value of profiles after filtering as the final label to be predicted. Due to the nature of sampling technique used by the VTune Profiler, it is possible that some functions do not have 10 profiling results before filtering, we remove these function. Finally, we are able to collect C function with reliable performance metrics from 62 C projects.

Step 4. C Function Extraction. We then extracted functions that have corresponding profiling results from the source code. We exploited Tree-sitter's Query API to find the specific function in the source file. We skipped functions without test cases and keep functions with profiling. Until now, we are able to construct $\langle function, profiling \rangle$ pairs.

Step 5. Dataset Construction. As we only extract functions with profiling data, after Step 4, we get 2,471 pairs of $\langle function, profiling \rangle$, where $profiling$ consists of four values. We split the constructed data samples into three sets: 80% of the samples (1,976 samples) are used for training, 10% are used for validation (247 samples), and the rest are used for testing (248 samples). The training set is used to adjust the parameters, the validation set is used to minimize overfitting, and the testing set is used only for testing the model with optimal parameters. Finally, the final model is used to predict CPU pipeline metrics online.

The distribution of the function length and CPU pipeline metrics is shown in Figure 6. We find that most C functions have less than 750 tokens and can be parsed into ASTs effectively. Besides, the four category metrics have different distributions. The *Front-End Bound*, *Back-End Bound*, *Retiring*, and *Bad Speculation* are distributed from 0%-40%, 0%-60%, 0%-80%, and 0%-30%, respectively.

5 EVALUATION

5.1 Baselines

As this is the first work on the on-the-fly pipeline metrics prediction task during the development before compiling, we use four baselines to demonstrate the effectiveness of our proposed model PGProf. We compare it with the following several baselines:

Mean Predictor. Similar to [14], we also consider using a *mean predictor*. As its name suggests, the mean predictor will take the mean value of each of the four categories in the training set as its predictions, irrespective of what input function is given.

Random Forest. Random Forest [7] is widely applied throughout software engineering studies (e.g., defect prediction [28] and security bug report prediction [48]) and shows robust and high performances. We exploit Bag-of-Word (BOW) to extract features from the source code. We leverage the CountVectorizer [9] as the BOW model to convert source code to vectors. Then, we use them as the feature vectors of the random forest. Each decision tree in the random forest learns to predict four metrics. All of the predictions of all the decision trees will be averaged to get the final prediction.

LSTM. LSTM [22] is one of the most popular recurrent neural network models. It has been widely applied to software engineering studies [4, 44]. LSTM is effectively to learn the sequential representation of code tokens. We use the LSTM to learn the code sequence and predict the profiling of the given code. For fair comparison, we use the same vocabulary and dimension size as the PGProf. The LSTM is a Recurrent Neural Network (RNN) and allows previous outputs to be used as inputs while having hidden states. The last hidden state is usually used as the representation of the input sequence [33]. Thus, we extract the last hidden state as the representation of the source code. Finally, we use the same MLP layers to predict pipeline values as PGProf.

Transformer. Transformer [42] is an encoder-decoder framework that has been successfully adopted in machine translation tasks. In this paper, we use the Encoder of Transformer to learn the code representation and predict the CPU profiling distribution according to the representation. Similar to the LSTM, we leverage the same hidden size and vocabulary as the PGProf for fair comparison. Transformer is based on the self-attention mechanism and each token is learnt by calculating attentions with other tokens. To get the representation for the input sequence, we leverage the element-wise maximal embedding of input tokens as the final representation.

5.2 Evaluation Metrics

In this study, PGProf predicts CPU pipeline values for C functions. To identify whether PGProf can predict these values or not, we leverage two metrics to evaluate the effectiveness of PGProf.

We introduce each measure as follows:

- *AbsoluteError*: We first report the absolute error between predicted values and target values:

$$AbsoluteError = \frac{1}{N \times 4} \sum_{j=1}^{N} \sum_{i=1}^{4} |\widetilde{v_i} - v_i| \tag{9}$$

[9]https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

Table 2. The detailed information of build tools

Tools	Cmake	GNU Make	Autoreconf	Meson	Ninja	GCC
Versions	3.16.3	4.2.1	2.69	0.62.2	1.8.2	9.4.4

where N represents the number of samples in the test set, and $\widetilde{v_i}$ and v_i refer to the prediction value and target value for pipeline metrics (i.e., *Retiring*, *Back-End Bound*, *Front-End Bound*, and *Bad Speculation*). All of the target values in the test set are measured through VTune. The $AbsoluteError$ is a less-is-better metric.

- $Acc@k$: is used to measure the accuracy within $k\%$ error. Different from the classification task, pipeline prediction is hard to be evaluated by traditional metrics, such as accuracy, recall, and F1 score. Thus, we propose to calculate the accuracy within a small deviation:

$$Acc@k = \frac{1}{N \times 4} \times \sum_{j=1}^{N} \sum_{i=1}^{4} \begin{cases} 0 & if\ |\widetilde{v}_i - v_i| > k\% \\ 1 & if\ |\widetilde{v}_i - v_i| \leq k\% \end{cases} \tag{10}$$

Specifically, we measure accuracy within 5% and 10%, i.e., $Acc@5\%$ and $Acc@10\%$. The $Acc@k$ is a higher-is-better metric.

5.3 Experiment Settings

We implemented PGProf in Python using the Pytorch framework[10] and the DGL library[11]. The hidden states of the graph nodes are 768 dimensions. It is trained using AdamW [34] with shuffled mini-batches. During the training process, the batch size is set as 16 and the gradient accumulation steps are set as 2. The model is trained as most 200 epochs with the early stop strategy. We evaluate on the validation set twice a epoch. If the model does not improve 16 times in a row, we will stop training in advance. As for the dropout [40], learning rate, unroll steps $\mathcal{T}$ and stacked layers count, we leverage the Optuna [3] to automate hyperparameter optimization. The search space of each hyperparameter is configured within a reasonable range to maximize the capability of Optuna. For example, we explore the dropout rate between 0.1 and 0.9, and for the learning rate, we investigate values ranging from 1e-4 to 1e-1. Note that we also use the learning rate scheduler to make the model converge faster. To get a fair comparison, we also apply automatic hyperparamter optimization to baselines but with different search space. We keep the best model with highest $Acc@5\%$ on the validation set. Our experiments are conducted on Ubuntu v20.04.1 64bits, with a RTX3090-24GB GPU and an Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz. The detailed information of build tools used in this paper is shown in Table2:

6 RESULTS

To gain a deeper understanding of the performance of our approach, we conduct an analysis of our evaluation results in this section. Specifically, we focus on three research questions:

- How effective is PGProf compared to the baselines?
- How effective is each component of PGProf?
- How efficient is PGProf?

6.1 RQ1: The Effectiveness of Our Approach

In this RQ, we want to investigate how effective our approach is and how much improvement our approach can achieve over the baselines. We apply our approach and the baseline methods (i.e., Mean Predictor, Random Forest,

[10]https://pytorch.org/.
[11]https://dgl.ai.

Table 3. Comparisons of PGProf with each baseline in terms of *AbsoluteError* and *Acc@k*. (*, **, *** and **** refer that the corresponding p-value is less than 0.05, 0.01, 0.005, and 0.001)

Model	*AbsoluteError*	*Acc* @5%	*Acc* @10%
Mean Predictor	0.088****	34.98%	66.13%
Random Forest	0.078****	42.94%	73.79%
LSTM	0.080****	43.15%	70.36%
Transformer	0.074***	46.17%	73.69%
PGProf	**0.067**	**49.90%**	**79.44%**

Table 4. Comparisons of PGProf with each baseline in terms of *AbsoluteError* (represented as *AE*) and *Acc@k* on each category.

Category	PGProf			**Mean Predictor**			**Random Forest**			**LSTM**			**Transformer**		
	AE	*Acc* @5%	*Acc* @10%	*AE*	*Acc* @5%	*Acc* @10%	*AE*	*Acc* @5%	*Acc* @10%	*AE*	*Acc* @5%	*Acc* @10%	*AE*	*Acc* @5%	*Acc* @10%
Front-End Bound	0.057	59.27%	83.06%	0.087	29.44%	64.92%	0.073	41.94%	75.40%	0.072	47.58%	72.58%	0.062	55.65%	80.24%
Bad Speculation	0.053	57.26%	89.11%	0.064	45.56%	82.26%	0.058	55.65%	85.08%	0.065	49.19%	79.84%	0.058	54.44%	85.89%
Back-End Bound	0.077	44.76%	72.18%	0.096	34.27%	61.29%	0.086	39.52%	70.56%	0.089	39.92%	65.73%	0.085	37.50%	67.74%
Retiring	0.081	38.31%	73.39%	0.104	30.65%	56.05%	0.094	34.68%	64.11%	0.094	35.89%	63.31%	0.091	37.10%	60.89%

LSTM, and Transformer) on the collected dataset, and compare their performance in terms of *AbsoluteError* and *Acc@k*.

Table 3 shows the results of our approach PGProf and the baseline techniques. We observe that PGProf outperforms all baselines in terms of all evaluation metrics. Among all approaches, Mean Predictor achieves worst performance on pipeline metrics prediction in general.

For *AbsoluteError*, all approaches can predict metric values within 10% error. PGProf can effectively predict these metrics within 7% error, i.e., it achieves *AbsoluteError* of 0.067. Using Transformer, the best performing model for comparison, our approach PGProf can reduce *AbsoluteError* by 7%. This result indicates that PGProf is capable of predicting pipeline metrics with less deviation in practical use. For *Acc@k*, PGProf achieves *Acc*@5% and *Acc*@10% of 49.9% and 79.44%, respectively. Using all baselines for comparison, these results constitute improvements of 8.0%-42.7% and 7.8%-20.1% in terms of *Acc*@5% and *Acc*@10%.

In addition, we also report the results of all approaches on each category shown in Table 4, including *Front-End Bound*, *Bad Speculation*, *Back-End Bound*, and *Retiring*. Among all metrics, all approach perform best on *Bad Speculation* in terms of *AbsoluteError*. As for *Acc*@5%, PGProf performs best on predicting *Front-End Bound* metrics, while other approaches performs best on *Bad Speculation*. We observe that all approach perform worst on *Retiring* metrics prediction except Mean Predictor, which has lower accuracy on *Front-End Bound*. PGProf has a deviation of 0.081 and the accuracy on it is 38.31% and 73.39% in terms of *Acc*@5% and *Acc*@10%. According to Figure 6, the distribution of *Retiring* is much more scattered from 0% to 80%. It is difficult for these techniques to predict such scattered distributed values.

To make the evaluation more comprehensive, we also try to vary the k in the *Acc@k* metric. As shown in Figure 7, the X-Axis is the specific k value, and the Y-Axis is the corresponding *Acc@k* value. In this type of figure, the closer the curve is to the upper left corner, the better the model is. We can observe that the PGProf beats all baselines by a considerable margin. Besides, we notice that the Transformer model beats the Random Forest

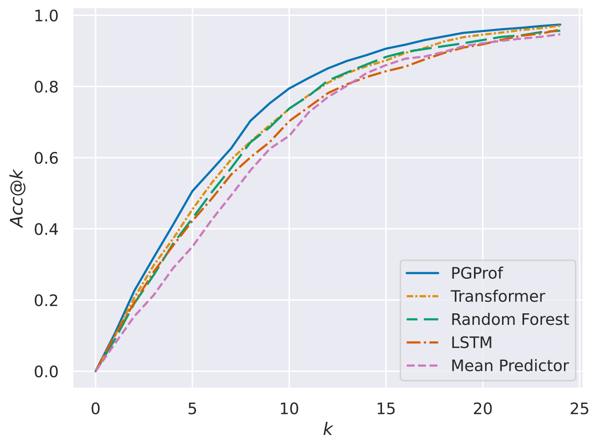


Fig. 7. Different approaches with varying k

model in *Acc*@5. However, the advantage decreases as the value of k increases. The Mean Predictor performs worst in nearly all possible values of k, but when the value of k is greater than 11, its performance is slightly better than that of the LSTM model. All approaches achieve nearly 100% accuracy when the value of k gets closer to 25. That is reasonable because the performance gap will be eliminated when the tolerance (e.g., k) becomes larger. We also conduct a hypothesis testing experiment to test whether the performance between PGProf and baselines is significant. First, we use the Shapiro-Wilk test to test if the *AbsoluteError* values are normally distributed. The result indicates it is not the truth. We then apply Wilcoxon signed-rank test [47] to check the significance at the confidence level of 95%. The results show that our approach significantly outperforms baselines. We choose the Wilcoxon signed-rank test because it is a non-parametric test and does not assume normality. The additional benefit is that it checks paired differences and the *AbsoluteError* between compared models is a paired structure.

In summary, PGProf outperforms the four baselines by a large margin. The *AbsoluteError* of PGProf indicates that it can help developers predict pipeline metrics within 0.067 deviation in On-the-Fly profiling.

6.2 RQ2: Ablation Study

In this paper, we use the graph neural network to learn different structural representations of C functions, including AST, CFG, and DFG. We want to investigate the impacts of these components on the performance of our approach. To illustrate the importance of each component, we compare our approach with five incomplete variants:

- *AST* predicts pipeline values by using the GGNN to learn only from the AST without other types of edges.
- *AST + NCS* adds the NCS edges into the AST.
- *AST + NCS + CFG* predicts pipeline values by removing the egdes from the DFG. We compare PGProf and it to measure the improvements from DFG.
- *AST + NCS + DFG* is similar to the above variant and used to measure the improvements from CFG.
- *AST + CFG + DFG* removes the NCS edges and uses them to measure the improvements from the NCS.

The effectiveness of the five variants are demonstrated in Table 5. All our variants outperform baselines (shown in Table 3) and prove the effectiveness of learning from structure information to predict profiling metrics. We observe that only using AST performs worst among all variants. Adding NCS, CFG, and DFG edges is helpful for PGProf to capture the code structures, thus improving PGProf's ability to predict performance metrics. When

Table 5. Effectiveness of each incomplete variant of our approaches in terms of *AbsoluteError* and *Acc@k*. (standard deviation σ is represented in the parentheses, and *, **, *** and **** refer that the corresponding p-value is less than 0.05, 0.01, 0.005 and 0.001)

Model	*AbsoluteError*	*Acc*	
		@5%	@10%
AST	0.073 (0.0425)***	47.68%	74.70%
AST + NCS	0.069 (0.0392)	46.47%	77.32%
AST + NCS + CFG	0.069 (0.0397)*	49.40%	76.82%
AST + NCS + DFG	0.070 (0.0414)*	49.50%	75.81%
AST + CFG + DFG	0.069 (0.0415)	49.09%	76.61%
PGProf	**0.067**	**49.90%**	**79.44%**

Table 6. Time costs of different approaches

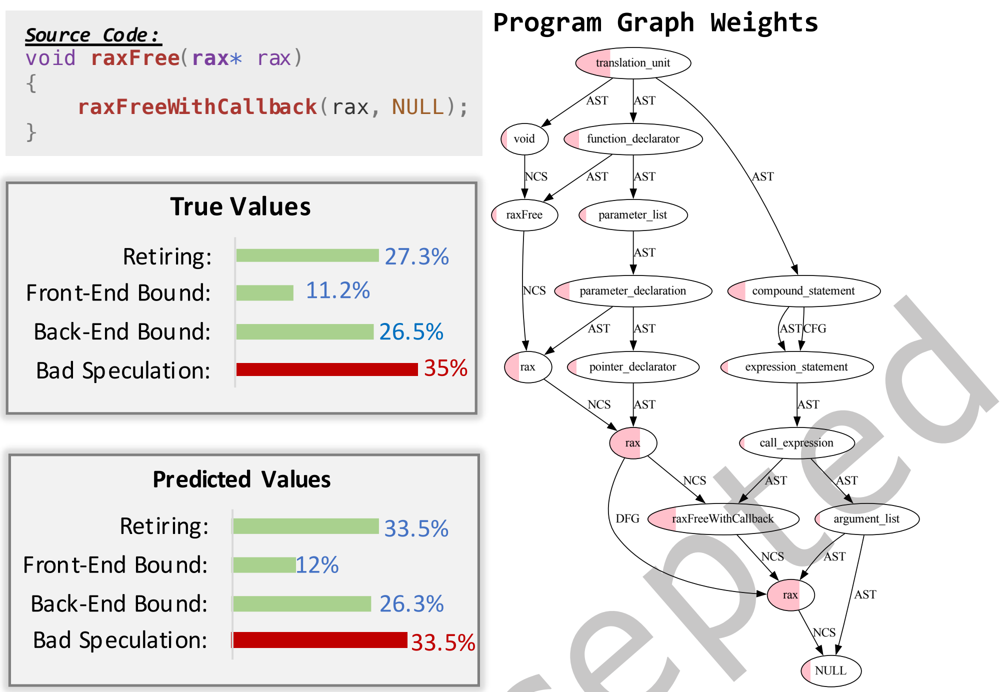
Approaches	Train	Test	Test One	Params
Random Forest	1.75m	0.029s	0.0001s	-
LSTM	8.88m	1.865s	0.0050s	44.8M
Transformer	11.36m	3.100s	0.0125s	45.5M
VTune	-	-	10.0000s	-
PGProf	6.40m	12.123s	0.0490s	194.6M

the NCS is added into the AST, the *AbsoluteError* decreases and improves accuracy. When integrating the CFG or DFG, we can find that they contribute more on the *Acc@K* and achieve improvements on *Acc@K*. It demonstrates that combining the AST, NCS, DFG, and CFG can make full use of its superiority. To get more insights, we also add the standard deviation in the parentheses. A similar hypothesis testing procedure was conducted to test the level of significance. The results are shown in Table 5. We can observe that PGProf outperforms AST, AST+NCS+CFG, and AST+NCS+DFG significantly. However, the increase is not significant when only integrating NCS or graphs into ASTs. It demonstrates the importance of the combinations of NCS and graphs in predicting programs' performance.

6.3 RQ3: Time Costs of our Approach

Neural network models need to be trained before being adapted to predict profiling metrics. The training process is conducted offline and can be used to make predictions online. In this research question, we want to investigate the training time cost and the test time cost of our approach to better understand the practicality of our approach PGProf. To measure the time complexity of our approach and other baselines, we record the start time and the end time of their training process and the test process. Note that we also include the time taken to translate the code into graph representation. For a fair comparison, all models are trained on the same machine containing an NVIDIA GeForce RTX 3090 GPU with 24 GB memory.

Table 6 illustrates the time costs of our approach and baselines. Random Forest is a machine learning based approach and does not have trainable parameters. Besides, it only takes 1.75 minutes to build the model and predicts values in less than one millisecond. Compared to deep learning based baselines, our approach PGProf



Source Code:
```
void raxFree(rax* rax)
{
    raxFreeWithCallback(rax, NULL);
}
```

Fig. 8. An example with high accuracy by learning from DFG

has the most parameters with about 194.6M trainable parameters. However, the training time is less than the LSTM and the Transformer. It takes about 0.0490s to predict performance values for a C function.

As for the traditional profiling approach VTune, it get the profiling information by sampling. In this paper, we run a test continuously for 10 seconds to get the CPU metrics by sampling. Even ignoring the time to compile and build, it takes more time than PGProf to get the profiling information. Compared to traditional profiling techniques that need to build, run, and monitor the software in the run time, our approach PGProf is much more efficient for practical uses.

7 DISCUSSION

7.1 What does PGProf learn from the program graph?

To better understand what PGProf learns from the program graph, we calculate the graph weights and the first two authors manually inspect the test results from collected C projects in GitHub, a total of 248 samples from the testing set. More specifically, we retrieve attention scores for all the nodes within a graph. Each attention score is converted into the red area in each node. The relative size of the red area corresponds to the importance of the node. In this way, we can clearly know which nodes the models' prediction has the greatest relationship with. Figure 8 demonstrates an example with low *AbsoluteError* and high accuracy. According to the graph weights, we can find that PGProf pay more attention on the data flow of the variable `rax`. In addition, the function call node `raxFreeWithCallback` also has high weights. Considering the high weights of the function call and the hot spot in *Bad Speculation*, we can infer that there may be a performance bottleneck in this function call of `raxFreeWithCallback`.

Table 7. Comparisons of PGProf with baselines in terms of *AbsoluteError* and *Acc@k* on -O2 Optimization.

Model	*AbsoluteError*	*Acc*	
		@5%	@10%
Mean Predictor	0.098	35.28%	59.68%
Random Forest	0.082	40.87%	69.71%
LSTM	0.092	35.42%	61.54%
Transformer	0.092	35.42%	62.18%
PGProf	**0.081**	**40.87%**	**70.03%**

Figure 9 shows another example that PGProf achieves high accuracy. PGProf can help developers to draw their attention to potential problem areas, in this case, to the high percentage of *Front-End Bound*. By analyzing the program graph weights, we can find that PGProf has similar attentions on different `if` branches. In addition, we observe that the pointer has higher weights. Multiple `c->flags` operations may cause the performance issues.

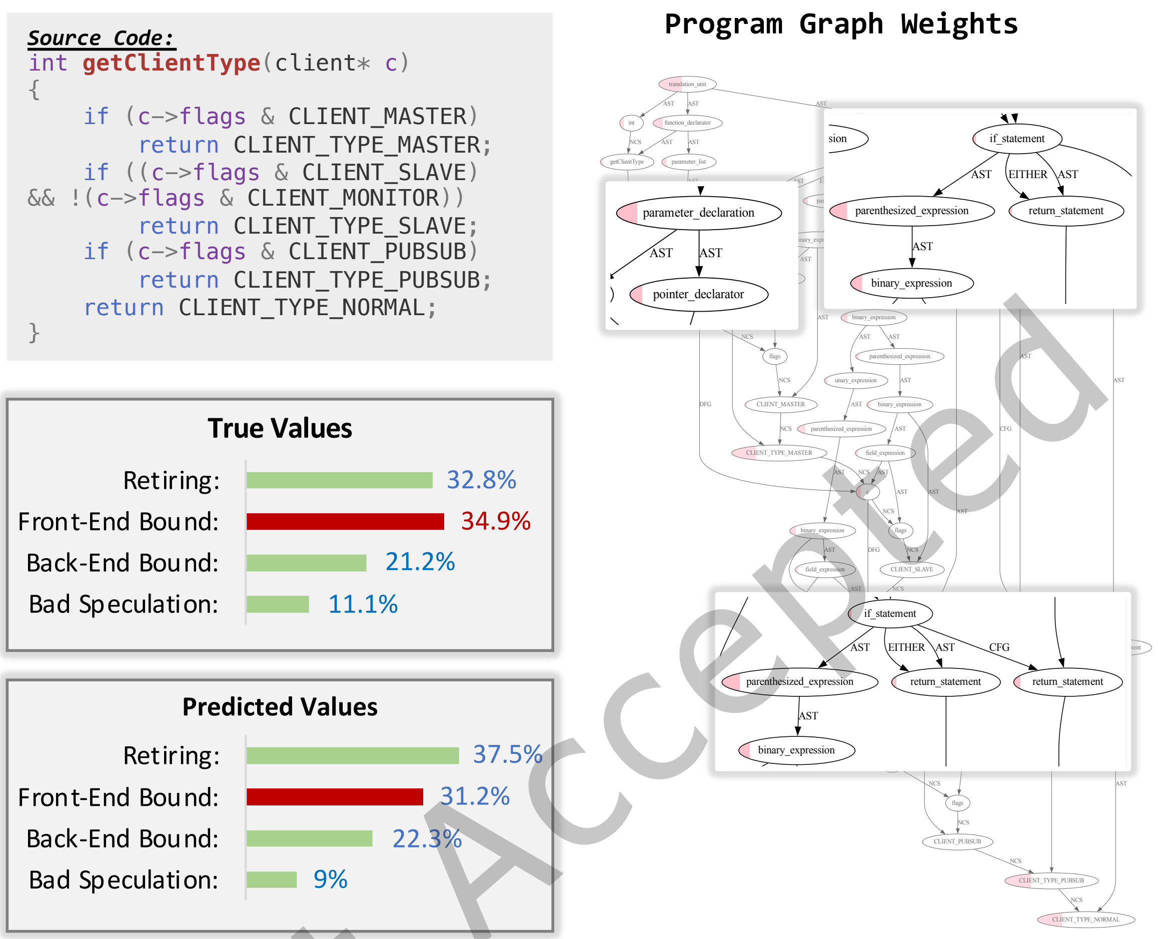
Based on our inspection, we summarize four major advantages of PGProf: First, PGProf can effectively learn the complex code structure from the source code, such as data flows, control flows, and function calls. As Figure 8 shows, PGProf can capture data flow information from `rax` variable. Second, it can learn the pointer usage. Pointer is one of the most important features of C programming language. Third, our program construction relies on the AST, and the different type of edges are added based on the C grammar provided by the Tree-sitter. It will be useful for code that does not yet compile, or contains errors, as the Tree-sitter is a robust parser that would provide useful results even in the presence of syntax errors. Learning the usage of C pointers in the source code can be used to predict performance for C functions. Fourth, PGProf can help developers to narrow down the scope of performance debugging. By extracting the nodes' weights in the readout layer, we can better understand how the model makes its prediction. Because our program graph is based on the AST, the problematic nodes can be easily mapped back to the corresponding parts of the origin C function.

7.2 When does PGProf fail?

We also investigate why our approach fail. Figure 10 depicts an example with low accuracy. We can find that PGProf grossly underestimate the *Bad Speculation*. PGProf does not learn the return statement in the `do-while` as the low weights in the return statement shown in Figure 10. The return within the `do-while` may cause the program exit earlier. Therefore, speculatively prefetching is likely to stall, leading to high Bad Speculation. However, PGProf gives high weights for the return statement at the end of the program which means PGProf consider the `do-while` will be executed and exit at the last return statement.

7.3 Compiler Optimizations

To analyze the impact of the compiler optimization, we random select five project and conduct an experiment on -O2 level. The results are shown in Table 7. We can observe that the performance of all approaches decreases in different degrees. Among them, PGProf achieves the best performance and the Random Forest is similar to PGProf. During the -O2 optimization, the semantic of the source code is abstracted, thus, the performance declines.



Source Code:

```
int getClientType(client* c)
{
    if (c->flags & CLIENT_MASTER)
        return CLIENT_TYPE_MASTER;
    if ((c->flags & CLIENT_SLAVE)
&& !(c->flags & CLIENT_MONITOR))
        return CLIENT_TYPE_SLAVE;
    if (c->flags & CLIENT_PUBSUB)
        return CLIENT_TYPE_PUBSUB;
    return CLIENT_TYPE_NORMAL;
}
```

True Values

Retiring:	32.8%
Front-End Bound:	34.9%
Back-End Bound:	21.2%
Bad Speculation:	11.1%

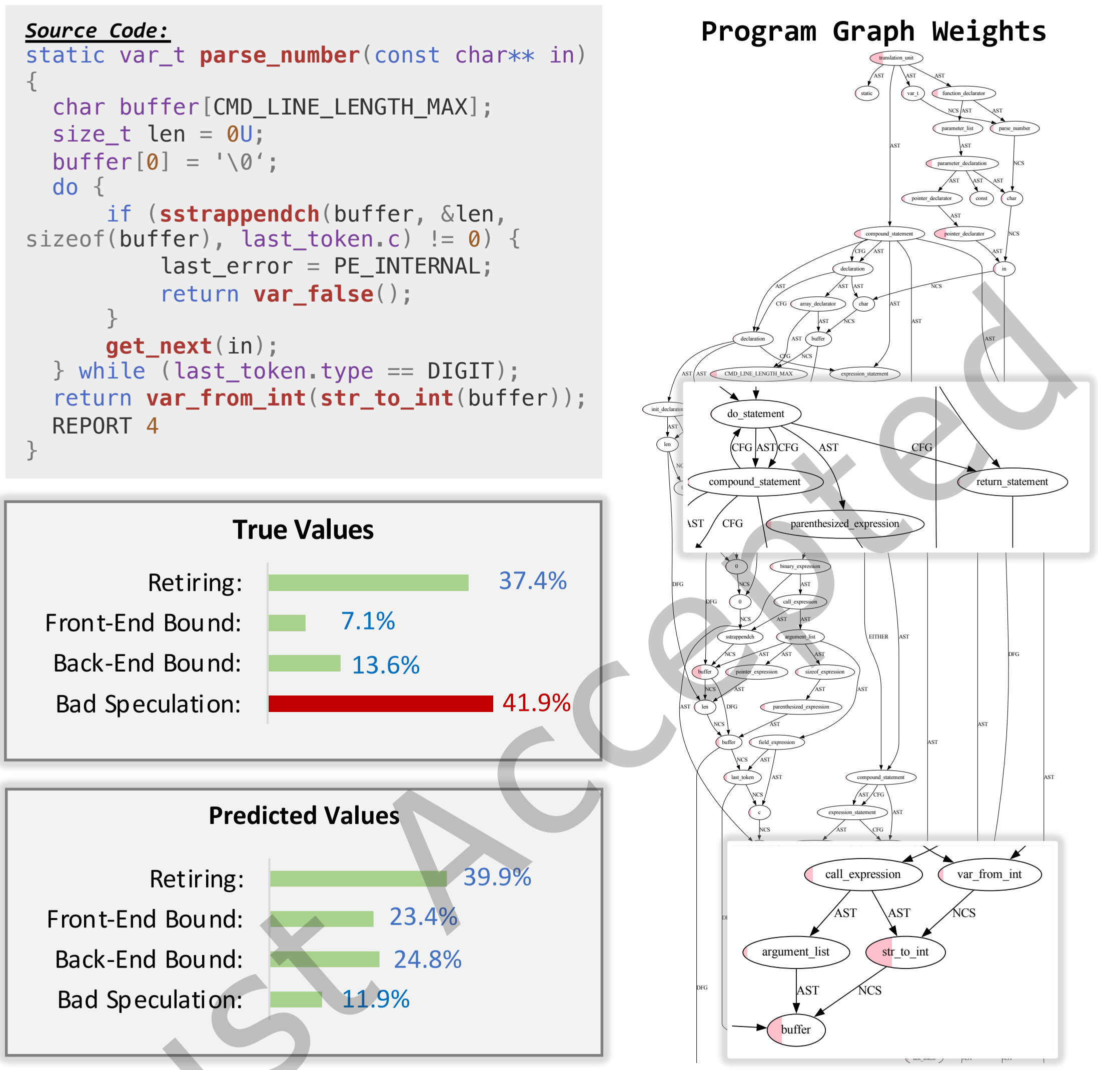
Predicted Values

Retiring:	37.5%
Front-End Bound:	31.2%
Back-End Bound:	22.3%
Bad Speculation:	9%

Fig. 9. An example with high accuracy by learning CFG

7.4 Transition to practice

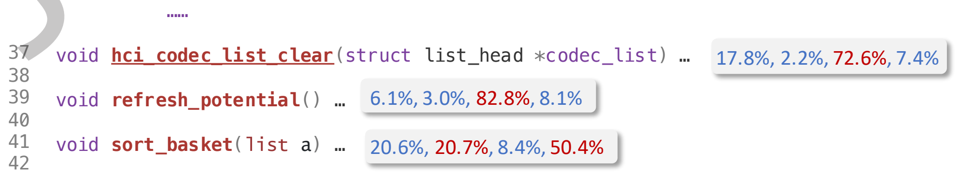
As mentioned above, we can integrate PGProf into an IDE to predict profiling metrics for programs on-the-fly. There are two possible usage scenarios. First, we can display the profiling data for a specific program (as shown in Figure 1). This process can be triggered by the developers, e.g., clicking the method name and selecting profiling. The time cost of this procedure is 0.0490s on average (shown in Table 6). Second, we can integrate the PGProf to analyze the whole file with multiple human-written methods. As shown in Figure 11, for each method, we analyze the source code and predict the profiling data. The time cost of the procedure depends on the number of methods N, i.e., $0.0490 * N$. The file-level analysis helps developers find the performance issues among different methods. To make it more practical, we may need to use other techniques, such as model distillation and distributed computing, to speed up model inference.



Source Code:

```
static var_t parse_number(const char** in)
{
  char buffer[CMD_LINE_LENGTH_MAX];
  size_t len = 0U;
  buffer[0] = '\0';
  do {
      if (sstrappendch(buffer, &len,
sizeof(buffer), last_token.c) != 0) {
          last_error = PE_INTERNAL;
          return var_false();
      }
      get_next(in);
  } while (last_token.type == DIGIT);
  return var_from_int(str_to_int(buffer));
  REPORT 4
}
```

Fig. 10. An example that PGPROF predicts with low accuracy



```
......
void hci_codec_list_clear(struct list_head *codec_list) …   17.8%, 2.2%, 72.6%, 7.4%

void refresh_potential() …   6.1%, 3.0%, 82.8%, 8.1%

void sort_basket(list a) …   20.6%, 20.7%, 8.4%, 50.4%
```

Fig. 11. A usage example for file profiling with multiple methods.

8 THREATS TO VALIDITY

One threat to the validity of this work is that we predict pipeline values instead of directly identifying bottlenecks for the source code. Because different types of applications have different thresholds. For example, the client application and the database application have *Front-End Bound* bottlenecks if the *Front-End Bound* value greater than 10% and 25%, respectively [12]. In the future, we plan to extend our work to predict pipeline values and identify bottlenecks for specific domain programs.

The second threat comes from the bias in the profiling information we collected by VTune. According to the VTune, the bias is unavoidable due to the the nature of sampling methodology VTune takes. In general, the sampling methodology will not be able to provide 100% accurate data. To address this threat, we run each project 10 times to approximate more accurate data and reduce underestimates or overestimates. In addition, we also use the Modified Z-Scores to remove outliers. Besides the sampling problem, another issue may be related to the tests we use. The number of tests varies for each project. However, we assure that all available tests are utilized.

Another threat is that we predict CPU performance for C functions. Except for CPU performance, there are many other performance tasks, such as profiling the time costs in the source code. Our study is the first step to on-the-fly performance profiling and we will extend this work to highlight candidate statements that may cause performance issues in future work. In addition, we predict the metrics for Intel CPUs. Considering that the Intel CPU is the most widely used, the applicability of PGProf is broad. When applied to other types of CPUs, we can just tune our approach on the newly collected CPU metrics.

Finally, the threat comes from the microarchitecture differences among various CPU architects. The CPU pipeline metrics were first proposed and implemented by Intel [51]. However, this method could also be adapted to AMD processors [26] with subtle differences. The microarchitecture can affect what performance events the PMU can collect and how pipeline metrics are calculated. Besides, the microarchitecture also affects how the four categories of pipeline metrics can be split into smaller parts [26]. Nevertheless, the four categories of pipeline metrics remain unchanged. They are the most general parts in the pipeline metrics hierarchy [51], which will be less affected by the microarchitecture [26]. The reason is that all modern CPUs share a similar architecture, which can be logically divided into two parts: Front-End and Back-End. However, the exact values may fluctuate in different microarchitectures due to the differences in hardware resources, which means that we need to fine-tune our model when adapting it to another CPU with different microarchitectures.

As VTune Profiler has some limitations on other types of programming languages (e.g., Java) analysis [13], it is much more difficult for us to collect profiling information of them. Thus, we conduct experiments on the dataset of C projects collected from GitHub which may not be representative of all programming languages. Considering that our approach is language-independent, we can retrain PGProf on the program graphs of other programming languages, e.g., Java and Python. We argue that our approach can be easily adapted to other programming languages. We plan to spend more time collecting the profiling information for programs in other programming languages and try to extend our work to benefit more developers in the future.

9 RELATED WORK

9.1 Performance Analysis and Profiling

Performance profiling is a typical approach for performance diagnosis that automatically instruments code to measure the average computation time within functions in the code base [24]. Profiling analyzes the run-time behavior of a program execution with respect to CPU consumption or execution time and finds performance bottlenecks [13], such as, gprof [17] and VTune. It usually reports a list of functions in which time is spent

[12]https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[13]https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/code-profiling-scenarios/java-code-analysis.html#java-code-analysis_GUID-183BAE41-AD29-45CB-BB72-70446B1811EA.

during the execution [16]. Profilers have been shown an important tool in industry to debug software systems with respect to resource usage during the runtime [45]. Gprof [17] exploits the compiler to automatically insert code at the beginning and end of functions to measure function statistics. Huang and Zhu [24] propose a novel approach to analyze the hierarchy of statistics to identify the most likely performance issues. VTune is one of the most popular tools to collect performance metrics of software and also used in our work to construct the dataset. In addition, there are many profilers to pinpoint hotspots in programs, such as Perf [14], JProfiler [15], and IBM Health Center [16]. Perf is a tool that uses the performance counters subsystem in Linux to trace dynamic control flow and identify hotspots. Huong et al. [24] propose a deep sparse FNN to predict performance for highly configurable systems. IBM Health Center monitors the status of a running application and pinpoints hotspots in Java programs. It can help to optimize application performance and reduce system resource usage. Although profiling techniques can help to detect performance issues in the software, it must be used during the runtime. In this paper, we propose a new approach to predict metric values of the percentage of CPU's pipeline slots in each category before execution and give feedbacks during the development.

9.2 On-the-Fly Profiling

The most related study to our approach is *PerformanceHat* [10, 11]. *PerformanceHat* integrates production monitoring information directly into the source code view and shows execution time for programs. It utilizes information collected in state-of-the-art monitoring tools to build a performance model that is integrated in the developer workflow in the IDE. With *PerformanceHat*, software developers are enabled to identify and prevent performance issues faster when source code is augmented with monitoring data and developers receive immediate (i.e., near real time) feedback on code changes. Different from *PerformanceHat* that shows execution time, PGProf predicts CPU pipeline slots for developers. It helps developers to analyze the hotspots that takes the most CPU time. Some studies exploit static and dynamic analysis techniques to detect inefficiencies in programs. Beigelbeck et al. [6] exploit the static performance analysis that enables interactive reasoning about static performance properties in the code during the development process. Xu et al. [49] employ various static and dynamic analysis techniques to detect memory bloat by identifying useless data copying. Glider [12] generates tests that expose redundant operations in Java collection traversals. These studies can pinpoint redundant operations that lead to resource wastage. However, they do not tell the resource usage of programs while they are running. Different from them, we aim to predict the CPU pipeline utilization given the source code before execution.

9.3 Graph learning for code intelligence

In the early stage of automating software engineering research area, code is usually represented as a sequence of tokens. However, the code itself is highly structured rather than a flat token sequence. This intuition has driven researchers to seek better-structured code representations such as AST and graphs. There exist two popular methods to extract graph representation from code.

First, the graph representation is extracted by existing tools. For example, Cheng et al. [8] use the APT [38] and SVF [41] tools to extract CFG and VFG from source code respectively. These two graphs are fused into one graph and learned by the k-GNN [37] model to detect where a C/C++ function has vulnerabilities. Joern [50] is a platform that can be utilized to generate various kinds of graph representations. Hin et al. [21] use Joern to extract the Program Dependence Graph (PDG) and then use the graph attention network to learn this graph, which can be used to detect if a specific statement has a vulnerability in a C/C++ function. Second, the source code can be parsed to AST and then add specific types of edges to get graphs. For example, Allamanis et al. [5] extract

[14]https://perf.wiki.kernel.org/index.php/Main_Page
[15]https://www.ej-technologies.com/products/jprofiler/overview.html
[16]https://www.ibm.com/support/pages/java-sdk-monitoring-and-post-mortem

AST from C# code and add twenty different types of edges. Most edges are related to the data flow between variables, which can be utilized to perform VARNAMING and VARMISUSE tasks using the GGNN model. Wang et al. [43] add thirteen different types of edges on AST. Compared to Allamanis's work, they put more emphasis on modeling the control flow (e.g., treat the If statement and While statement differently). Then, this graph serves as input to GGNN and Graph Matching Network to do code clone detection tasks. Our work falls into the latter method, that is, we add different types of edges on AST based on our task. The construction of the program graph and the node encoding strategies in our work differ from existing approaches. Additionally, to the best of our knowledge, we are the first to apply graph neural networks to CPU pipeline metrics prediction.

10 CONCLUSION AND FUTURE WORK

This work aims to predict CPU pipeline metrics (including *Front-End Bound*, *Bad Speculation*, *Back-End Bound*, and *Retiring*) for developers during the development process. These metrics can help developers to understand how efficiently their code is passing through the core pipeline and determine hotspots in the source code in time. To tackle this task, we propose an approach named PGProf, which leverages a novel graph neural network model to learn from program graphs and predict pipeline metrics. Several improvements from different code structures, e.g., AST, CFG, and DFG, are introduced in PGProf to handle the characteristics of this task. Comprehensive experiments on a dataset with over 2,000 real-word C function and pipeline metrics samples show that PGProf outperforms three baselines by large margins and can reduce the efforts that developers perform for performance analysis during the development period. In the future, we plan to extend PGProf to predict the performance change trend according to the code change. We also plan to adapt PGProf to other programming languages, such as Java. In addition, it would be an interesting future direction to propose more advanced techniques to address PGProf's limitations.

ACKNOWLEDGMENT

This research is supported by the National Key R&D Program of China (No. 2024YFB4506400), CCF-Huawei Populus Grove Fund, and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 2015. Chapter 6 - Introduction to Profiling. In *Power and Performance*, Jim Kukunas (Ed.). Morgan Kaufmann, Boston, 105–118.

[2] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 1–12.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[6] Aaron Beigelbeck, Maurício Aniche, and Jürgen Cito. 2021. Interactive Static Software Performance Analysis in the IDE. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 490–494.

[7] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[8] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.

[9] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics, Doha, Qatar, 103–111.

[10] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C Gall. 2018. PerformanceHat: augmenting source code with runtime performance traces in the IDE. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 41–44.

[11] Jürgen Cito, Philipp Leitner, Martin Rinard, and Harald C Gall. 2019. Interactive production performance feedback in the IDE. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 971–981.

[12] Monika Dhok and Murali Krishna Ramanathan. 2016. Directed test generation to detect loop inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 895–907.

[13] Jiaqing Du, Nipun Sehrawat, and Willy Zwaenepoel. 2011. Performance Profiling of Virtual Machines. *SIGPLAN Not.* 46, 7 (mar 2011), 3–14.

[14] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, and Olivier Temam. 2007. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th international conference on Computing frontiers*. 131–142.

[15] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[16] Liang Gong, Michael Pradel, and Koushik Sen. 2015. Jitprof: Pinpointing jit-unfriendly javascript code. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 357–368.

[17] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 120–126.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[19] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).

[20] James R Hamilton et al. 2007. On Designing and Deploying Internet-Scale Services.. In *LISA*, Vol. 18. 1–18.

[21] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80.

[23] Erh-Wen Hu, Bogong Su, and Jian Wang. 2017. Software performance prediction at source level. In *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*. 263–270.

[24] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*. 76–91.

[25] Boris Iglewicz and David C Hoaglin. 1993. *Volume 16: how to detect and handle outliers*. Quality Press.

[26] Mateusz Jarus and Ariel Oleksiak. 2016. Top-Down Characterization Approximation based on performance counters architecture for AMD processors. *Simulation Modelling Practice and Theory* 68 (2016), 146–162.

[27] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: platform-independent memory debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 345–356.

[28] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[29] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 249–260.

[30] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of ICLR'16*.

[31] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[32] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[33] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 585–597.

[34] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[35] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. 2021. Learning to Make Compiler Optimizations More Effective. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 9–20.

[36] Hao Miao, Jiazun Chen, Yang Lin, Mo Xu, Yinjun Han, and Jun Gao. 2023. JG2Time: A Learned Time Estimator for Join Operators Based on Heterogeneous Join-Graphs. In *Database Systems for Advanced Applications: 28th International Conference, DASFAA 2023, Tianjin, China, April 17–20, 2023, Proceedings, Part I*. Springer, 132–147.

[37] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. 2019. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 4602–4609.

[38] Keshav Pingali and Gianfranco Bilardi. 1995. APT: A data structure for optimal control dependence computation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. 32–46.

[39] Yangyang Shu, Yulei Sui, Hongyu Zhang, and Guandong Xu. 2020. Perf-AL: Performance Prediction for Configurable Software through Adversarial Learning. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 11 pages.

[40] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[41] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

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

[43] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[44] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[45] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-Box Performance-Influence Models: A Profiling and Learning Approach (Replication Package). In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 232–233.

[46] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[47] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 196–202.

[48] Xiaoxue Wu, Wei Zheng, Xin Xia, and David Lo. 2021. Data quality matters: A case study on data label correctness for security bug report prediction. *IEEE Transactions on Software Engineering* (2021).

[49] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–430.

[50] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[51] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. 35–44.

[52] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.

[53] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.