Static Hot Path Prediction using Advanced Deep Learning

Zirui Zhao
zhaojer@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Xinyun Cao
xinyunc@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Jiawei He
jiaweihe@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Jiwei Liu
ljwdre@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Hanzhe Guo
hanzheg@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Abstract

Hot path identification is crucial for tasks like code optimization and debugging. The traditional approaches of dynamic profiling require full program execution and introduce runtime overhead. We draw inspiration from previous work that used deep learning methods to perform hot path prediction statically and evaluate modern transformer models for this task. We evaluated 1) a fine-tuned BERT model and 2) a GPT with prompt engineering against a benchmark data set. Our results show that the fine-tuned transformer model can perform hot path prediction better than deep learning models proposed by previous work.

Keywords

LLVM, Intermediate Representation, Compiler Optimization, Hot Paths, Machine Learning

1 Introduction

Understanding and optimizing program execution behavior is crucial for modern software development and compilation. A well-established observation in program analysis is that applications spend the majority of their execution time in a small subset of possible execution paths, commonly referred to as "hot paths". This insight, documented by Ball and Larus [1], has profound implications for program optimization, as identifying and optimizing these frequently executed paths can yield significant performance improvements.

The traditional approach to identifying hot paths is through dynamic profiling, which involves instrumenting and executing the program to collect runtime behavior data [1]. While this method provides precise information about path execution frequencies, it comes with several significant drawbacks. First, it requires executing the entire program with representative inputs, which may be difficult to obtain or time-consuming to generate. Second, the profiling process itself introduces runtime overhead and can significantly slow down program execution. Finally, in rapid development environments where code changes frequently, the need to re-run dynamic profiling after each modification can create an impractical bottleneck in the development cycle.

These limitations have motivated research into static approaches for predicting hot paths without program execution. The CrystalBall system demonstrated the feasibility of using deep learning, specifically recurrent neural networks (RNNs), to predict hot paths by analyzing program structure at the intermediate representation (IR) level [9]. However, the field of deep learning has advanced significantly since CrystalBall's introduction, with transformer-based models achieving unprecedented success in sequence processing tasks [8]. These advancements suggest an opportunity to improve upon existing static hot path prediction techniques.

In this work, we explore the application of modern transformer models to the hot path prediction problem. Specifically, we investigate two approaches: (1) fine-tuning BERT, a bidirectional encoder model known for its strong performance in sequence classification tasks, and (2) prompt engineering with GPT, a state-of-the-art large language model. We evaluate these approaches on a dataset created from the PolyBench benchmark suite [6], comparing their performance against each other and contextualizing our results with respect to previous work. Our research questions focus on: (1) whether modern transformer models can effectively predict hot paths from program IR, (2) how different transformer architectures compare in this task.

For reproducibility and further research, all source code for this project is made publicly available on GitHub, while the dataset and the model are available on HuggingFace (see Appendix A for specific links).

2 Related Work

The task of predicting program execution behavior, like path-based behavior, has been previously studied. CrystalBall [9] represents a significant milestone in using deep learning for static prediction of hot paths. The authors introduced a novel technique that leverages recurrent neural networks (RNNs) to identify frequently executed paths by analyzing LLVM intermediate representation (IR). Unlike prior approaches that relied on source code or manually crafted features, CrystalBall operates directly on compiler IR, making it language-independent. Their RNN-based approach achieved an AUROC of 0.85 on SPEC CPU2006 benchmarks, demonstrating the feasibility of using deep learning for static path prediction.

Another notable approach in static program analysis is presented by Brauckmann et al. [2], who explored using graph neural networks (GNNs) to learn from compiler-based representations like abstract syntax trees (ASTs) and control-data flow graphs (CDFGs). Their work shows that graph-based representations can capture important structural properties that may be lost in sequential representations. In contrast to CrystalBall's sequence-based approach, their GNN models demonstrated superior performance on tasks like heterogeneous OpenCL mapping.

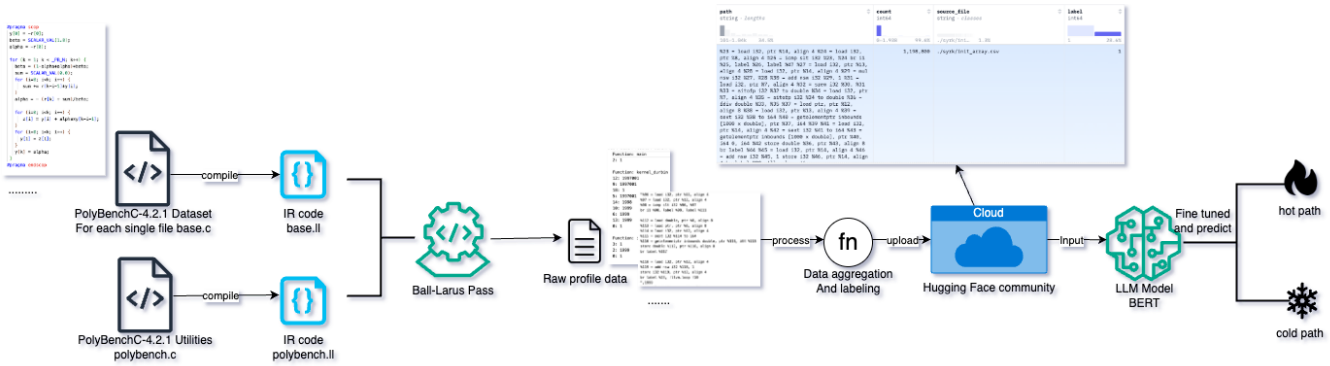

Figure 1: Overall Study Procedure.

Static prediction of loop behavior has also been studied by Tetzlaff and Glesner [7], who proposed using machine learning, specifically Random Forests, to predict loop iteration counts. Their work focused on enabling hot spot optimizations by automatically learning relationships between static code features and dynamic behavior, achieving high accuracy across different benchmark suites.

Our work builds upon these foundations but takes a different approach by using modern transformer models (BERT and GPT) to learn path patterns. While CrystalBall showed the effectiveness of RNNs for this task, we explore whether the strong sequence understanding capabilities of transformers can provide even better results. Unlike Brauckmann et al.'s graph-based approach, we maintain a sequential representation but leverage the transformer's self-attention mechanism to capture long-range dependencies.

3 Method

This study evaluates the feasibility and effectiveness of using a fine-tuned BERT model and prompt-engineering a pre-trained GPT model to predict hot paths in code execution. First, we implemented the Ball-Larus dynamic path profiling algorithm to accurately identify hot and cold paths in programs. We then applied this algorithm to the PolyBench Benchmark Suite to generate the dataset for training and testing. Subsequently, we fine-tuned the BERT model and employed prompt-engineering on the pre-trained GPT model, assessing their performance using metrics including accuracy, AUROC, and F1 score. Each step of the experiment procedure, illustrated in Figure 1, is comprehensively discussed in the following sections.

3.1 Ball-Larus Path Profiling

The Ball-Larus path profiling algorithm, introduced in 1996, is a seminal technique for efficiently profiling program execution paths [1]. The algorithm assigns unique numbers to all possible paths through a procedure's control flow graph and instruments the code to track which paths are executed at runtime. The key insight of Ball and Larus was that by carefully choosing the edge weight values in the control flow graph, each unique path can generate a unique path number through simple arithmetic operations during program execution.

The algorithm works by first converting the control flow graph into a directed acyclic graph (DAG) by removing back edges. It then assigns weights to edges such that the sum of weights along any path from entry to exit produces a unique number. This is accomplished by visiting nodes in reverse topological order and computing weights based on the number of paths to the exit node. The instrumentation is then strategically placed only on certain edges to minimize runtime overhead while still capturing full path information.

While the Ball-Larus algorithm is well-documented in literature, we were unable to find an open-source implementation that works with modern compiler infrastructures. Therefore, we developed our own implementation targeting LLVM 18, which we have made publicly available (link provided in Appendix). Our implementation follows the original algorithm while taking advantage of LLVM's modern intermediate representation and optimization capabilities.

Figure 2 illustrates the complete workflow of our path profiling implementation. The process begins with a C source file (foo.c) which is first compiled to LLVM IR (foo.ll). Our Ball-Larus pass then analyzes this IR, implementing the path numbering algorithm and inserting the necessary instrumentation code. This instrumented code includes calls to our runtime library which maintains path execution counts. After linking with this runtime library, the final executable is produced.

When the instrumented program executes, it generates a profile output file containing path execution counts for each function, with paths identified by their Ball-Larus assigned path IDs. Since these numeric IDs are not directly meaningful to our machine learning models, we developed a path regeneration program (regen) that converts these IDs back into their constituent sequences of basic blocks and instructions. The conversion requires additional metadata about the program's control flow graph (edge, basic blocks and number of paths) which is also output by our instrumentation pass.

The final output is a set of CSV files where each record contains two fields: the sequence of LLVM IR instructions comprising the path (with basic blocks and instructions delimited by newline characters), and the execution count for this path.

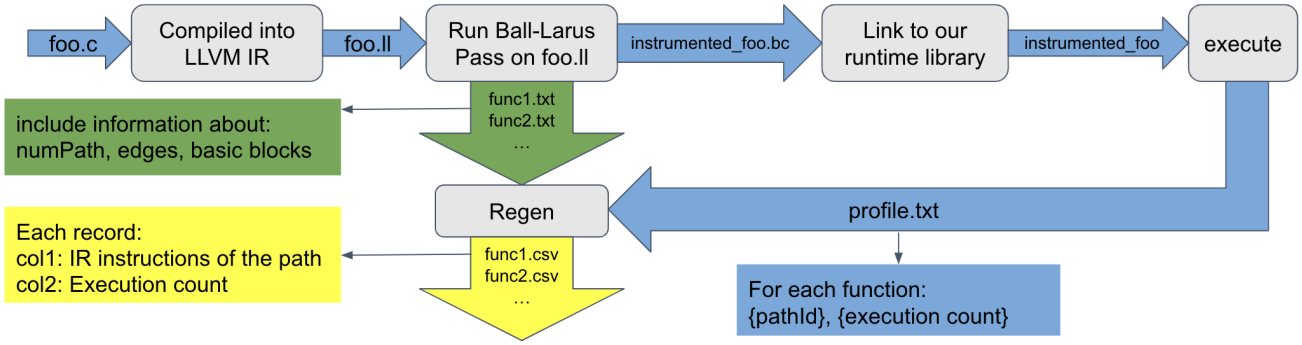

Figure 2: Ball-Larus Path Profiling Pipeline: From Source Code to Dataset.

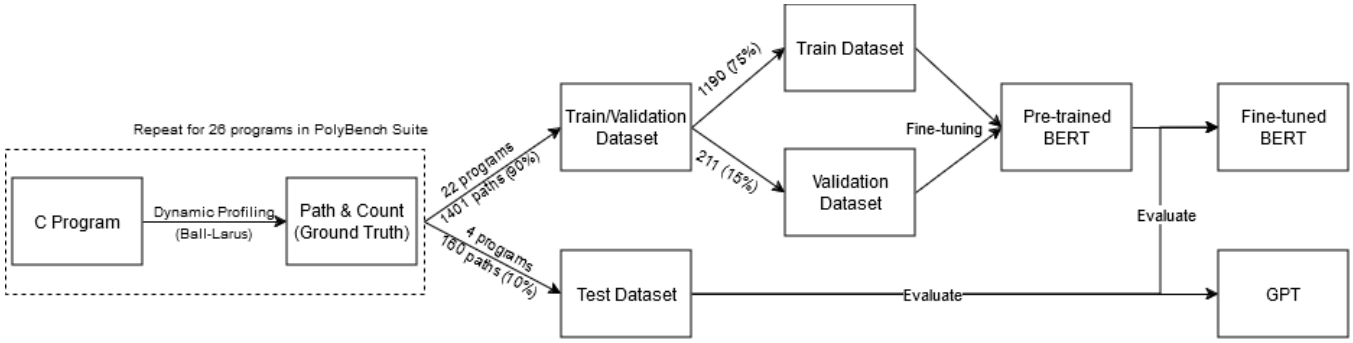

Figure 3: Dataset Creation Procedure.

3.2 Dataset

3.2.1 Benchmark & Dataset Creation. We utilized PolyBench [6], a benchmark suite of 30 computationally intensive C programs extracted from various application domains, as the source programs for our dataset. PolyBench was selected for its simplicity and compatibility with our study due to the following reasons:

- Each program in PolyBench is self-contained, written in a single .c file accompanied by a corresponding .h file, and independent of other programs. This allowed each program to be compiled and analyzed individually.
- The programs are concise, typically between 100 to 200 lines of code, providing sufficient complexity for path profiling without being unwieldy.
- PolyBench covers diverse application domains, including linear algebra, image processing, and physics simulations, offering a representative sample of real-world programming scenarios.
- Its open-source and free nature aligned with our project's constraints, as it operates without external funding.

To create the dataset, we applied our Ball-Larus path profiling implementation to all PolyBench programs, capturing the paths and execution counts for each. During compilation to LLVM intermediate representations, four programs (deriche, cholesky, gramschmidt, correlation) encountered linker errors that we were unable to resolve, resulting in their exclusion. Consequently, path profiling was conducted on the remaining 26 programs, yielding a total of 1561 paths. This profiling process spanned over five hours, emphasizing the computational cost of dynamic profiling and highlighting the potential advantages of leveraging machine learning for static profiling.

Paths were labeled as "hot" (1) if executed at least once (execution count ≥ 1) and "cold" (0) otherwise, following the definition used in CrystalBall [9]. This produced an imbalanced dataset with 468 hot paths (30%) and 1093 cold paths (70%), reflecting the natural predominance of cold paths in real-world programs. For each path, we also recorded its associated program/function (e.g., durbin/kernel_durbin.csv) to facilitate potential future analyses requiring program-specific information.

3.2.2 Dataset Splitting. To create training, validation, and test sets, we adopted a hierarchical splitting strategy designed to ensure model evaluation integrity. We first split the dataset into training/validation and test sets by programs, ensuring that paths from a single program do not appear in both sets. This was essential to prevent data leakage, as paths from the same program might share similarities that could artificially inflate model performance if present in both training and test sets.

For test set selection, we performed stratified random sampling at the program level, using program domains (e.g., linear algebra) as strata. This ensured a representative distribution across domains while maintaining an approximate 10% share of total paths and

preserving the hot-to-cold path ratio. The resulting test set comprised paths from four programs (jacobi-2d, syr2k, durbin, and 2mm), totaling 160 paths (10%).

The remaining 22 programs were further split into training and validation sets using `scikit-learn`'s [5] standard splitting method while preserving the exact hot-to-cold path ratio. The final training set contained 1190 paths (75%), and the validation set included 211 paths (15%). A detailed breakdown of dataset paths is presented in Table 1. The overall dataset creation process is illustrated in Figure 3.

Table 1: Paths in Split Dataset.

	Training	Validation	Test
Hot Paths	340	60	68
Cold Paths	850	151	92

3.2.3 Dataset Usage. The training and validation sets were employed for fine-tuning the BERT model, while the test set was reserved for evaluating both the fine-tuned BERT model and the pre-trained GPT model.

3.3 Models

3.3.1 BERT. We employed BERT [3], a pre-trained transformer model renowned for its ability to process sequences with a deep contextual understanding. As an encoder-only model, BERT is particularly well-suited for classification tasks, which aligns with the focus of this study. Additionally, the base BERT model comprises 110 million parameters, making it one of the smaller pre-trained transformer models, which translates to faster inference times and a reduced ecological footprint, further supporting its selection for this research.

To fine-tune BERT, we utilized Hugging Face's state-of-the-art `Transformers` library [8], built on the PyTorch framework. The process began with tokenization using the `BertTokenizer` API, which converts raw text—in this case, LLVM IRs of paths—into tokenized sequences that the model can process. The base BERT model has a maximum sequence length of 512 tokens, meaning it can process only 512 unique "words" per input and truncates anything longer. While some paths in our dataset contain over 1,000 instructions, an analysis revealed that none of the paths exceeded 512 unique tokens. Thus, a potential loss of information was not a concern in this study.

We then initialized the pre-trained BERT (base, uncased) model using the `BertForSequenceClassification` API, configured for binary classification with two labels (0 for "cold path" and 1 for "hot path"). The model outputs a predicted class label and an associated probability score between 0 and 1, representing its confidence in the prediction.

We fine-tuned BERT for 3 epochs on the training/validation datasets using the `Trainer` API, with standard hyperparameters including a batch size of 16 and a learning rate of 5e-5 (additional details can be found in the source code linked in the Appendix). Fine-tuning was executed on Google Colab's NVIDIA T4 GPUs and took approximately 30 minutes. We also monitored key metrics including loss and accuracy after each epoch to avoid overfitting.

3.3.2 GPT. In addition to fine-tuning BERT, we also experimented with GPT, specifically the GPT-4o model [4], as a zero-shot predictor for hot paths. Unlike BERT, which was specifically fine-tuned on our dataset, GPT relied solely on its extensive pre-training on large-scale, general-purpose textual data. Our goal was to investigate whether the state-of-the-art large language model could understand and predict hot paths from LLVM IR sequences without any additional training or domain adaptation.

We prompted GPT-4o with carefully crafted system and user messages. The system message instructed the model to output only a single decision—indicating whether the given path is hot or not—and a confidence score between 0 and 1. Specifically, the model was asked to produce one of the following lines:

- `"This is a hot path, confidence: X"`
- `"This is not a hot path, confidence: X"`

Here, X is a floating-point number representing the model's confidence. We then converted these responses into probabilities that a path is hot. For `"This is a hot path"`, we took X directly as the probability. For `"This is not a hot path"`, we used $1 - X$ to define the probability that the path is hot. This simple scheme allowed for straightforward evaluation using the same metrics as with BERT, enabling a fair comparison.

We applied GPT-4o directly on the test set, adhering to the zero-shot setting, i.e., without providing any examples from the dataset. The chosen test set paths had not been previously seen by GPT and consisted solely of LLVM IR representations. The inference process was carried out through OpenAI's API [4], which required no local GPU resources. While this approach is computationally expensive and slow due to API calls, it offers an efficient way to assess GPT's performance without additional fine-tuning.

3.4 Evaluation Metrics

To assess the performance of our models, we employed three evaluation metrics: accuracy, area under the receiver operating characteristic curve (AUROC), and F1-score, measured on the test dataset.

Accuracy represents the proportion of correctly classified paths among all paths in the test set. It serves as a straightforward and intuitive measure of overall performance. However, accuracy can be misleading for imbalanced datasets, as it may be disproportionately influenced by the majority class (cold paths in our case).

AUROC evaluates the model's ability to distinguish between hot and cold paths across all possible classification thresholds. It measures the trade-off between the true positive rate (sensitivity) and false positive rate, providing a robust metric for imbalanced datasets. A high AUROC indicates that the model consistently assigns higher probabilities to true hot paths compared to cold paths, regardless of the threshold.

F1-score is the harmonic mean of precision and recall, capturing a balance between the two. It is particularly valuable for imbalanced datasets, as it accounts for both false positives and false negatives. F1-score provides insight into how well the model identifies hot paths (positive class) without excessive misclassification.

By incorporating these metrics, we aimed to achieve a comprehensive understanding of model performance. These metrics were computed using `scikit-learn`'s [5] evaluation utilities with a default threshold of 0.5.

Table 2: Performance Comparison between Models

	RNN (CrystalBall)	BERT (fine-tuned)	GPT (prompt-engineer)
Accuracy	N/A	0.98	0.44
AUROC	0.85	0.99	0.54
F1 score	0.82	0.99	0.54

4 Results & Analysis

The results presented in the Table 2 clearly demonstrate that fine-tuning modern transformer-based models, specifically BERT, significantly outperforms both the RNN-based method from prior work and the zero-shot performance of GPT-4o.

Fine-tuned BERT achieved an AUROC of 0.99 and an F1-score of 0.99, indicating exceptional performance in distinguishing hot paths from cold paths in LLVM IR sequences. These near-perfect scores highlight BERT's ability to effectively learn and adapt to the domain-specific patterns present in the dataset. In contrast, GPT-4o performed poorly, with an AUROC of 0.54 and an F1-score of 0.54. These results, close to random guessing, suggest that GPT-4o struggled to leverage its pre-trained knowledge in a zero-shot setting for this specialized classification task. Meanwhile, the RNN-based model from CrystalBall [9] achieved an AUROC of 0.85 and an F1-score of 0.82 on SPEC CPU 2006 benchmarks. Although direct comparisons are not possible due to differing datasets, the substantial gap in performance highlights the promising potential of fine-tuning recent transformer models for hot path prediction.

BERT's exceptional performance can be attributed to its encoder-only architecture, which is specifically optimized for classification and sequence understanding tasks. Its bidirectional attention mechanism allows it to capture intricate patterns in LLVM IR sequences, making it well-suited for hot path prediction. Additionally, fine-tuning on the LLVM IR dataset enabled BERT to adapt its pre-trained knowledge to the structural and semantic characteristics of LLVM IR paths, further enhancing its performance.

In contrast, GPT-4o's decoder-based architecture is primarily designed for generative tasks like text generation, which may not align as well with binary classification. Moreover, GPT-4o relied entirely on its pre-training data, which likely did not include LLVM IR-like syntax or semantics, leaving it poorly equipped to handle this domain-specific task. Since LLVM IR sequences possess a structure and vocabulary distinct from the natural language or general-purpose code GPT-4o was trained on, this creates a mismatch between the model's knowledge and the task requirements.

Another factor contributing to BERT's near-perfect performance may be the homogeneity of the dataset. Although the test set excluded programs seen during training, all programs were sourced from the PolyBench benchmark suite, which might share structural similarities. This could have made the task easier for BERT, raising questions about its generalizability. Future studies could address this by evaluating the model on entirely different datasets, such as SPEC, to better understand its robustness.

To reiterate, while our fine-tuned BERT model achieved superior results compared to CrystalBall's RNN-based method, it is important to note that the comparison is indirect due to differences in datasets and evaluation setups. CrystalBall's results were derived from SPEC benchmarks, while our study focused on PolyBench programs. Nonetheless, these findings highlight the potential of transformer-based models, particularly when fine-tuned on domain-specific data. The significant improvement in AUROC from 0.85 (RNN) to 0.99 (BERT) suggests that transformer architectures, combined with fine-tuning, represent a promising direction for advancing hot path prediction. This finding underscores the importance of leveraging recent advances in natural language processing for program analysis tasks.

5 Future Work

5.1 Dataset

While our work demonstrates the potential of detecting hot paths using machine learning models on datasets like PolyBench. A critical step forward would involve testing our approach on larger and more diverse benchmarks, such as the SPEC CPU benchmark suite. SPEC CPU is widely regarded as a standard for evaluating compiler optimizations and provides a comprehensive set of real-world applications across diverse domains. This would allow us to assess the scalability and robustness of our proposed methods in handling more complex and varied IR structures. Due to time constraints and the sheer scale of SPEC CPU, we were unable to include it in our experiments for this study. Preparing and profiling such a large dataset requires more computational resources than expected and careful preprocessing, potential work including rewriting program of profiling execution paths and generating annotations for machine learning tasks. In future work, we propose extending our pipeline to handle the SPEC CPU dataset. This effort would help validate the generalizability of our approach to industrial-scale applications.

5.2 Few-shot Training on LLM

Another promising method is the utilization of GPT-based models for few-shot learning. Though our study primarily focuses on zero-shot classification. Maybe a few-shot paradigm could enhance the model's understanding of LLVM IR code. Few-shot training involves providing the model with a limited number of annotated examples during inference. In future work, we plan to develop a systematic framework for few-shot training using models like GPT-4 or other Generative LLMs. By using high-quality examples of hot and cold paths along with corresponding labels, we can guide the model to better distinguish execution patterns with minimal additional data. This approach would also reduce dependency on extensive labeled datasets.

5.3 Potential usage of LLM Compiler

The success of transformer-based architectures in natural language processing and their adaptation to code-related tasks highlights their potential for LLVM IR analysis. Future work will focus on exploring transformers trained specifically for LLVM IR, such as the LLM Compiler built by Meta and experimenting with pre-trained transformer models like CodeBERT or GraphCodeBERT. This opens up a promising direction for improving hot path detection. The LLM Compiler is trained on an extensive corpus of LLVM IR, x86_64 assembly, and ARM assembly in order to understand and optimize

IR code. Though the compiler is primarily intended for tasks like code size optimization and disassembling assembly into IR, it can be adapted to the problem of hot path detection. The model's tokenizer is fine-tuned on LLVM IR syntax. Therefore it can process instruction-level semantics effectively. This capability ensures that IR paths can be represented as tokens and passed into the model for processing. The LLM Compiler then generates dense embeddings for tokenized input. These embeddings can serve as a foundation for downstream tasks such as classification or regression which are essential for hot path detection. These proposed models could be used to capture both the semantic and structural aspects of LLVM IR paths. Therefore, fine-tuning pre-trained LLMs represents a promising direction, particularly if sufficient funding is secured to support the computational resources required for such tasks. Fine-tuning also would allow us to adapt large models to our specific problem domain.

5.4 IR2Vec

While IR2Vec provides a robust method for generating embeddings from LLVM Intermediate Representation (IR) at the program, function and instruction levels, it is not directly applicable to our proposed method for hot path detection due to its current limitations. First limitation is granularity. IR2Vec operates at the program level or function level meaning that it only supports embedding entire IR files or functions into fixed-length vectors. However, our task requires embeddings at the path level (sequences of basic blocks), or at the block level (basic blocks). Another limitation is the reliance on the whole program context. IR2Vec relies on the global structure of the program to generate embeddings so the vector representation of an instruction or block is influenced by the entire program. In contrast, our approach requires embeddings that can capture the semantics of a path or block independently of program-level context to distinguish isolated hot/cold paths in the dataset. A possible method could be extending IR2Vec to generate path-level or block-level embeddings and integrating dynamic profiling data into the pipeline; we can build a machine learning framework tailored for hot path analysis. This future work points out a possible advancement in compiler optimization research. Promoting finer-grained analysis and facilitating new opportunities for dynamic code optimization could be a future direction for IR2Vec.

6 Conclusions

In this paper, we used a fine-tuned BERT model and prompt-engineered GPT models to perform hot path prediction and compared the results with prior work. The evaluation results with the PolyBench benchmark suite dataset show that fine-tuning modern transformer-based models outperforms RNN-based methods from previous work and zero-shot performance of GPT-4o. We discussed the implications and limitations of our approach. We also proposed future work, including diversifying the dataset, performing few-shot training on LLM, and using an LLM Compiler and IR2Vec.

References

[1] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 46–57. https://doi.org/10.5555/243846.243857

[2] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2019). https://doi.org/10.48550/arXiv.1810.04805.

[4] OpenAI. 2024. ChatGPT-4o. https://platform.openai.com/docs/overview Accessed: 2024-11-18.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[6] Louis-Noel Pouchet and Tomofumi Yuki. 2016. PolyBench. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1 Accessed: 2024-11-18.

[7] Dirk Tetzlaff and Sabine Glesner. 2013. Static Prediction of Loop Iteration Counts Using Machine Learning to Enable Hot Spot Optimizations. In *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, Los Alamitos, CA, USA, 300–307. https://doi.org/10.1109/SEAA.2013.12

[8] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[9] Stephen Zekany, Daniel Rings, Nathan Harada, Michael A. Laurenzano, Lingjia Tang, and Jason Mars. 2016. CrystalBall: Statically Analyzing Runtime Behavior via Deep Sequence Learning. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. IEEE Computer Society, Washington, DC, USA, 130–142. https://doi.org/10.1109/CGO.2016.32

A Project Source Code

The source code of this project is publicly available on the GitHub Organization: cse583

This organization contains 3 repositories:

(1) cse583/ball-larus: source code for the Ball-Larus path profiling algorithm and the regen program to generate the dataset in the format that can be processed by the models
(2) cse583/transformers: source code for creating the dataset and fine-tuning/evaluating the BERT model for hot path prediction
(3) cse583/gpt: source code for prompt-engineering and evaluating OpenAI's GPT-4o model for hot path prediction

The dataset and the fine-tuned BERT model are publicly available on Hugging Face at the following repositories respectively:

(1) zhaojer/compiler_hot_paths
(2) zhaojer/bert-hot-path-predictor