ML-LOOP: Optimizing LLVM with Machine Learning

Anurag Bangera, Chirag Bangera, Jonhan Chen, and Richard Wang
University of Michigan, Ann Arbor, MI
{abangera, cbangera, asclkose, riwa}@umich.edu

***Abstract*—We present a comprehensive approach to using multiple machine learning models to predict an optimized phase ordering given a predefined set of optimization passes. Phase ordering is in reference to the phase ordering problem, an ongoing issue in finding the optimal ordering to run selected optimization passes for a pre-compiled program in order to induce maximum speed up of the post-compiled code. An implementation of this algorithm was first defined by Fanfakh et al., 2020 [1]. In the original approach, a very limited subset of static features was used as training data for a machine learning model trained and tested on an even more limited set of data. In our implementation, we aimed to expand on the shortcomings of the previous implementation by tripling the number of collected features for models to be trained on through dynamic analysis and loop analysis, but also pair this with multiple machine learning models, including the K Nearest Neighbors model, Support Vector Machines model, Ada Boost, Gradient Boost, and Random Forest model. We also expanded the testing and training data fed into the model, allowing the model to be more effective when working with real-world code examples. The effect of this growth in scope led us to findings that suggested that although this idea shows promise for potential speedup of statically ordered optimization passes, the speed up found from the research paper on their data set was less than realistic and may not always in other training and testing data sets.**

I. Introduction

THE PHASE ORDERING PROBLEM can be defined as how to find the optimal way to order a set of optimization passes.

- Given an optimization pass OP1 and code base C1, unless the pass is redundant and fails to apply to the code it is optimizing, the pass will implement changes on the pre-compiled code, thus changing C1 to an edited version C2.
- The next optimization pass OP2 will be optimizing a different code base C2, thus potentially leading to different modifications to code base C2 than if it had been optimizing C1.

This demonstrated interdependence between optimization passes is extremely hard to predict is varies between different code bases, thus generally a static approach to taken to pass ordering. However, due to this, there is theoretically a dynamic pass ordering for every code base that outperforms the current static approach taken to defining a pass ordering. Thus, the benefit of creating a dynamic ordering is clear: an opportunity to gain further speed up of post-compiled code with the same optimization passes.

The Machine Learning LLVM Ordering of Optimization Passes (ML-LOOP) dynamically tailors the pass orderings based on features found and collected within the code and trains a series of models in an attempt to predict and define a more efficient ordering. The process of doing so is two-fold: Firstly, the given code is analyzed for features within three main categories: Static Features, Dynamic Features, and Loop-Based Features. Secondly and finally, the features are used as training for 3 machine learning models: K Nearest Neighbors, Support Vector Machines, and Random Forest. The trained models are thereafter fed testing data and return a dynamically generated pass-ordering permutation.

A. Feature Collection

Features are statistical inputs for the machine learning models, and thus are generally relevant to the information used during optimization passes, as gathering features in this fashion allows the models a higher likelihood of discerning patterns found within optimization passes.

1) Static Features: To collect static features, a static analysis was run on the code. From this, multiple features were able to be discerned, including but not limited to the following: The counts of the number of ALU, memory, and Branch instructions of the static code were found.

2) Dynamic Features: To collect dynamic features, a dynamic analysis was run on the code. The dynamic instruction counts of the ALU, Memory, and Branch instructions were found, as well as the ratio of those static to dynamic instructions.

3) Loop Features: In order to collect loop features, all loops were analyzed in the program, and from those, the total amount of loops, as well as the average amount of Basic Blocks per loop were counted. Within each loop, the instruction counts of ALU, Memory, and Branches were also counted. In addition to this, branch analysis was conducted on the programs, counting the total amount of biased and unbiased branches.

B. Machine Learning Models

Using the features described above, 5 models were used on the training data gathered: K Nearest Neighbors, Support Vector Machines, and Random Forest. A random number generator was used as a control group to test the classifier against a random phase ordering selection.

1) K Nearest Neighbors: The 'K' closest data points were used in order to predict the next best optimal pass in the phase order.

2) Support Vector Regression (SVR): Finds a function that matches the form of the specified kernel that approximates the relationship between input variables and a continuous target variable. The three different kernel functions we tested were: Linear, Poly, and RBF.

3) Random Forest: Uses decision trees during training to output a result in the test data sets.

4) Gradient Boost: It builds models in a stage-wise fashion, constructing new models and taking the combined weighted sum to make a final prediction.

5) Ada Boost: Builds a classifier and continually adjusts the parameters based on the training data.

II. Related Works

A. Mitigating the Phase Ordering Problem Using Sub Sequences and Machine Learning

This paper details another approach to the phase ordering problem with machine learning, called MiComp [2]. Our research builds upon the MiCOMP framework, a pioneering approach that employs machine learning to mitigate the phase-ordering problem in LLVM optimizations. MiCOMP identified that O3 had 5 specific sub-sequences to permute between.

III. Implementation

The following section discusses our implementation of a solution to the phase ordering problem using machine learning models. It provides an overview of the features we chose to extract and their justifications, a discussion on how the data set and models were created, as well as an overview of the workflow of the implementation.

A. Feature LLVM Passes

This subsection focuses on the feature extraction aspect of the implementation. It will go into depth as to what features were extracted using LLVM passes as well as the justification behind each feature selected. Three passes were written to support this project:

1) A ***function pass***, extracts high-level features from each function of the program it is run on.
2) A ***loop pass***, extracts features from each loop of the program it is run on.
3) A ***module pass***, extracts the static instruction counts from the program it is run on and is used to compare our results to the features used by our research paper benchmark.

1) Function Pass: The following section discusses what features were extracted in our function pass. The following features, along with their justification for selection are discussed below:

- **Total Basic Block Count**: The total number of basic blocks in a piece of code is crucial for measuring similarity, as it provides an overview of the code's structural complexity. A higher total basic block count suggests a more intricate code structure, potentially indicating complex algorithms or detailed procedural logic. A lower count implies a simpler, more linear code structure with fewer branching points. Comparing this metric helps identify similarities in overall code organization.
- **Average Instruction Count Per Basic Block**: Calculating the average number of instructions per basic block is valuable for comparing code similarity and complexity. A higher average instruction count per basic block suggests more operations within smaller code segments. A lower value implies simpler and more concise basic blocks. This feature helps in determining the level of detail in coding patterns and can be vital for distinguishing between high-level and low-level implementations.

Table 1: Instruction Categories

Instruction Categories
Integer ALU
Function Pointer ALU
Branch Instructions
Memory Instructions
Other

- **Dynamic Instruction Count Categories** Opcodes were divided into the categories found in Table 1. They were divided into these categories as they represent a well-rounded categorization for the purpose of an instruction. Dynamic instruction counts were collected as they capture the actual execution behavior of the code. Higher counts in specific categories indicate a greater emphasis on those operations during runtime, providing insights into the actual computational load and performance characteristics of the code. Analyzing dynamic instruction statistics is essential for identifying runtime characteristics and potential performance bottlenecks, contributing to a more accurate assessment of code similarities.
- **Static Instruction Count Categories**: Static instruction counts offer insights into the code structure without executing it. Higher counts in specific categories indicate a greater emphasis on those operations within the code segment, offering insights into the code's purpose and potential performance bottlenecks. Similarly to examining dynamic instruction statistics, comparing static instruction statistics provides robust insights into code complexity and similarities.
- **Dynamic To Static Instruction Count Category Ratios**: The dynamic-static ratios per instruction provide a new perspective on code behavior. Higher ratios suggest that certain instruction categories are more prevalent or executed more frequently during runtime compared to their static representation. Lower ratios may indicate that the static representation adequately captures the runtime behavior. This feature is crucial for understanding how much of the code's structure is reflected in its runtime execution, as well as providing insights into how often certain blocks and instructions are being run, helping to identify possible similar code sections.
- **Biased Branch Count**: Biased branches can significantly impact code execution efficiency. A higher biased branch

count suggests a more deterministic decision-making process within the code, potentially reflecting specific patterns or conditions that frequently lead to particular outcomes. Conversely, a lower count implies a more balanced set of branches without a strong bias towards true or false conditions. Measuring the bias in branch instructions helps identify common decision-making patterns, helping to identify similar conditional structures across different code segments.

- **Unbiased Branch Count**: Conversely, unbiased branches represent a more evenly distributed decision-making process. A higher unbiased branch count suggests a more balanced decision-making process within the code, potentially contributing to a clearer and more versatile control flow. A lower count implies a more biased set of branches, which may indicate more deterministic decision paths. Collecting data on unbiased branches aids in finding code segments with diverse control flow, contributing to the identification of code similarity.
- **Loop Count**: The number of loops in code is an indicator of its structure and efficiency. A higher loop count indicates a more iterative and potentially complex algorithmic design, whereas a lower count suggests a simpler, more linear code structure. Counting the number of loops helps to recognize repetitive patterns and helps identify what types of repeated operations the code is performing, making it a key feature for measuring code similarity.
- **Average Basic Block Count Per Loop**: Calculating the average number of basic blocks per loop provides insights into the complexity and operations of loop structures. A higher average basic block count per loop suggests more intricate decision-making and control flow within the loops, potentially indicating complex algorithmic structures. A lower value implies simpler loop structures with fewer basic blocks. This feature helps differentiate between simple and complex loop implementations, contributing to a more insightful comparison of code similarities.
- **Total Static Instruction Count In Loops**: Aggregating the static instruction count within loops highlights the computational load associated with looped sections. A higher total static instruction count in loops indicates significant loop involvement in the code. A lower value suggests simpler loop structures with fewer static instructions. This feature is important for identifying performance-critical loops and evaluating similarities in looped code segments.
- **Average Static Instruction Count Categories Per Loop**: The average static instruction count per loop offers a normalized view of the computational complexity within loop structures. A higher average static instruction count suggests that loops are involved in more intricate computations, potentially indicating complex algorithms or data processing. A lower value implies simpler loops. It helps in comparing the complexity and purpose of looped code segments, helping to identify code similarity.
- **Recursive Function Call Count**: Recognizing recursive structures is essential for understanding code architecture and purpose. A higher count indicates a greater reliance on recursive patterns, which can be crucial in algorithms or data structures that leverage recursion. A lower count suggests a less recursive or more iterative approach. Counting the occurrences of recursion provides valuable information for measuring similarity.

Running this pass results in a total of 30 features. The name of this pass is registered with the pass name of ***"fp"*** and can be used when loaded with ***opt***.

2) Loop Pass: The following section discusses what features were extracted in our loop pass. The following features, along with their justification for selection are discussed below:

- **Number Of Nested Loops**: The count of nested loops within a piece of code is a powerful metric for understanding its control flow complexity. A higher count indicates a more intricate nesting pattern, potentially signifying complex algorithms or deeply layered iterations. On the other hand, a lower count suggests a simpler, more linear control flow. This feature helps identify patterns of nested iterations, allowing for a powerful comparison of code segments based on their loop-nesting structures.
- **Number Of Outermost Loops**: The number of outermost loops, or parent loops, indicates the level of hierarchy in loop structures. A higher average loop depth may indicate a more intricate control flow, potentially signaling complex algorithms or data structures. Conversely, a lower value suggests simpler loop structures. This feature helps identify the nesting relationships between loops, providing insights into the overall organization of code and helping with the comparison of code segments with similar loop hierarchies.
- **Average Loop Depth**: Calculating the average depth of loops in code offers a normalized measure of loop nesting. This feature helps in comparing the overall depth of loop structures, by providing insights into code segments with varying levels of nested complexity.
- **Average Outer Loop Depth**: The average depth of outer loops in code provides a focus on the nesting structure of higher-level loops. This feature is valuable for assessing the overall hierarchy of loop structures within a code segment. By concentrating on outer loops exclusively, it offers a more targeted insight on the organization of primary iterations, allowing for a robust comparison between different code segments. A higher average outer loop depth indicates a more intricate control flow structure, while a lower value suggests a simpler loop hierarchy. This feature is beneficial for identifying similarities in the overarching loop organization of code, helping in the categorization and comparison of code segments based on their outer loop nesting complexity.
- **Deepest Loop Nest Depth**: The deepest loop nest depth measures the most extended chain of nested loops, providing a view of the code's structural complexity. This feature identifies highly intricate loop structures that may have a significant impact on code efficiency and readability. A deeper loop nest suggests a more

intricate and potentially challenging-to-understand code segment, while a shallower depth may indicate simpler loop structures. Analyzing the deepest loop nest depth helps in the identification of similarities based on the most complex nesting patterns present in different code snippets.

Running this pass results in a total of 5 features. The name of this pass is registered with the pass name of ***"lnp"*** and can be used when loaded with ***opt***.

In total, running these two passes results in the collection of 35 total features for our machine-learning models.

3) Module Pass: The following section discusses what features were extracted in our module pass. This module pass collects the static instruction counts of each opcode and is nearly identical to the instruction count pass used by [2]. There was a slight modification as the researchers were not transparent with some instruction count categories, so the instruction count of every possible opcode was collected resulting in 65 total features. This pass was created to benchmark our features.

B. Training Dataset Generation

This subsection focuses on the dataset generation aspect of the implementation. It will go through the steps done to generate the model training dataset for the machine learning model solution. All of the following scripts were written in Python.

1) Generate Pass Order Permutations: In total, there were 120 possible permutations of pass orderings based on the subsequences uncovered by [3]. A script was written to generate these permutations and will be leveraged in the next step. This is done with the script *generatePermutations.py*.

2) Best Optimization Pass: For each file in the training file directory, we run each optimization pass TODO times. Each optimization pass was assigned an identification number. After this, we selected the median time from all iterations for this specific pass. This median time was assigned to the corresponding passes' identification number. Once all passes for a specific file were run, the pass with the shortest time was chosen as the label for the file. The filenames and labels were all compiled into a CSV file. This is done with the script *runOptimizations.py*.

3) Feature Extraction: For each file in the training file directory, we run the function pass and loop pass discussed in the previous subsection. The filename and features were compiled into a CSV file. This is done with the script *collectFeatures.py*.

4) Training Dataset Creation: Upon completion of the previous steps, the CSVs were simply merged in order to create a training dataset with features and labels. This is done with the script *combineFeatures.py*.

C. Machine Learning Models

This subsection focuses on the machine learning aspect of the implementation. It will go through the steps done to train the machine learning model, apply the model to our test files, as well as create visualizations. All of the following scripts were written in Python.

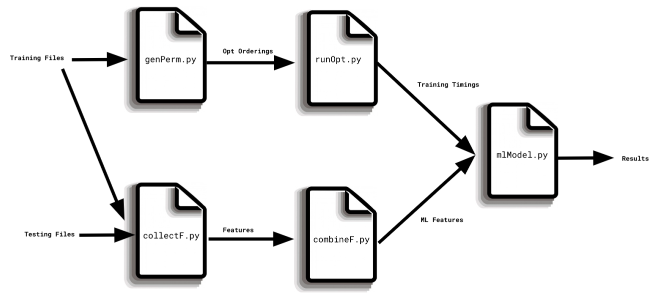

Fig. 1. A diagram depicting the workflow of generating the training dataset

1) Machine Learning Model Training and Prediction: The models are trained with the generated training dataset. A similar dataset is generated for all of the test files. The models leverage the features of the test files to make predictions and output a pass sequence identification number. These predictions are outputted in CSV format. This is done with the script *mlModels.py*.

2) Prediction Evaluation: The results of the models are then evaluated. Each test file is run with the corresponding predicted optimization pass alongside O1, O2, O3. The outputted execution times are in CSV format. This is done with the script *inference.py*

3) Visualization: The results of the models can be visualized with various graphs. The outputted graphs are in PNG format. This is done with the script *visualization.py*.

IV. Results

Model	Mean	Median
Random Forest	0.991403	2.992999
SVR Linear	11.056050	12.120381
SVR Poly	9.760627	13.818095
SVR RBF	4.510338	4.192152
KNN	3.245576	4.068354
Gradient Boosting	0.756240	4.068035
AdaBoost	0.101429	-0.025427
Random Number	-1.663757	-0.281425
O0	-4.147188	-2.940761
O1	-2.978962	-2.326182
O2	-4.065964	1.207549

Table 1: Performance Results of Model Predictions

A comparative analysis of our study's findings as shown in Fig 2 against the results documented in the original paper as shown in Fig 3 revealed noteworthy disparities and enhancements across multiple optimization models. Notably, our experiments demonstrated significant improvements in mean and median performance metrics when juxtaposed with the original paper's reported outcomes.

Among the evaluated models, SVR Linear, SVR Poly, and SVR RBF emerged as notable performers, showcasing substantial improvements over the baseline optimization levels (O0, O1, and O2). SVR Linear exhibited a noteworthy 11.06% increase in mean performance and a substantial 12.12% improvement in median performance compared to the baseline

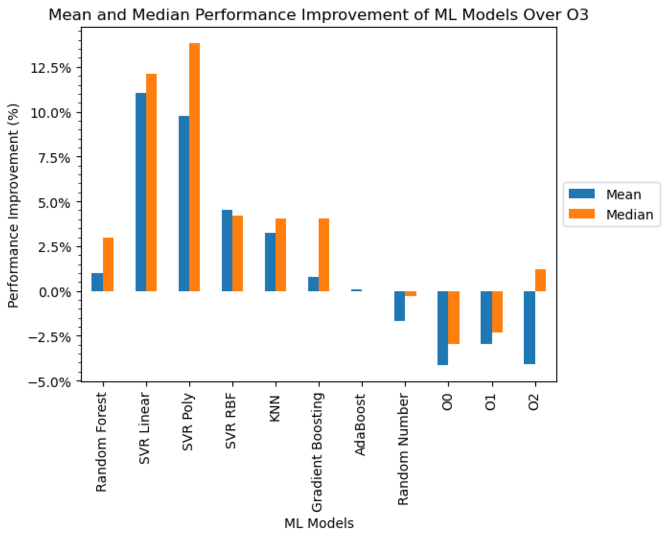

Fig. 2. Our New Feature Results

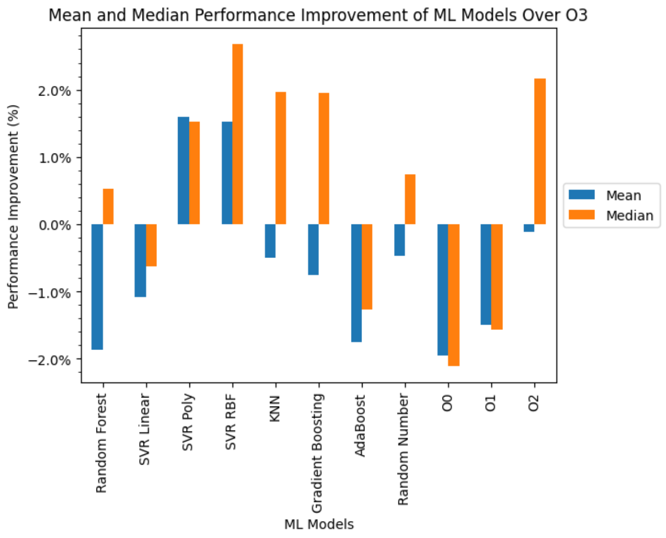

Fig. 3. Original Paper Feature Results

optimizations. Similarly, SVR Poly demonstrated a significant 9.76% increase in mean performance and an impressive 13.82% improvement in median performance. Additionally, SVR RBF displayed a commendable 4.51% enhancement in mean performance and a notable 4.19% boost in median performance relative to the baseline strategies.

While not achieving as high improvements, other models also displayed varying degrees of enhancement. KNN showcased a moderate 3.25% increase in mean performance and a 4.07% improvement in median performance over the baseline optimizations. Gradient Boosting exhibited a modest 0.76% increase in mean performance but displayed a similar 4.07% improvement in median performance. AdaBoost and Random Forest presented minor improvements over the baseline strategies, with AdaBoost showing a 0.10% increase in mean performance and a slight -0.03% change in median performance, while Random Forest displayed approximately 0.99% improvement in mean performance and a 2.99% increase in median performance.

In contrast, the utilization of a Random Number generator resulted in adverse effects, showcasing decreases of -1.66% in mean performance and -0.28% in median performance compared to the baseline O3 optimization. Comparatively, the baseline optimizations (O0, O1, and O2) demonstrated lower performance compared to the models, with O0, O1, and O2 displaying -4.15%, -2.98%, and -4.07% mean improvements, respectively, and varying median performance changes.

These results highlight the varying degrees of improvement across different models, underscoring the efficacy of SVR-based approaches, particularly SVR Linear, SVR Poly, and SVR RBF, in significantly enhancing performance compared to baseline optimization strategies.

V. Discussion

Results analysis here

VI. Future Works

While the results themselves show a small performance bump relative to baseline O3, we see that our current work has a lot of room for improvement and has carved out a very interesting path for future work in optimization. This proof of concept shows that features do in fact correlate with optimization performance. Using this fact, we can scale our current work with more test cases and increase the results. We see a lot of potential with the use of deep learning within this space. The true place this can be deployed is to analyze the relationships between different optimizations. By learning this, a program can then intelligently apply these optimizations. With large amounts of data, a potential training approach could use completely randomized, ungrouped pass orderings. Over hundreds of thousands of input files and training, we believe that a strong relationship can be established between the performant pass order and the features analyzed. A deep learning model on such a scale would be able to learn this relationship, as well as retrain and continuously learn over its deployment with the proper infrastructure. We believe that we have just scratched the surface of the performance gains that can be unlocked through this methods, potentially upwards of 10% per program.

Another potential future optimization is to reduce the granularity of our feature selection. We noticed that while collating feature results from different functions, a lot of the uniqueness of the functions themselves was lost. In addition, programs can be very large, so categorizing by features would 1. be difficult to gather training data, and 2. Not granular enough to judge relationships between features and pass ordering. Using function analysis within the workflow could better increase both the training efficiency and the model accuracy, helping the model achieve its ultimate SLO of performance. One of the biggest bottlenecks we faced in this project was the fact that profile data is inconsistent and hard to track between two relatively similar, small programs. This led to a decrease in the accuracy of our training heuristic, hurting our model. This method could potentially be much more effective for training models that aim to achieve other service-level objects that are deterministic, such as code size. This metric can be statically analyzed and would only require a single pass of our potential pass orders. This is a great future area of research with our training flow.

VII. Conclusion

Our study advances the understanding of the compiler phase-ordering problem by employing a diverse set of machine-learning models [4]. We expanded the feature set for model training, incorporating dynamic and loop analysis data, and tested this approach on a comprehensive dataset. While we achieved notable improvements in optimizing LLVM compiler performance, our results also highlighted the inherent challenges and variability of outcomes across different codebases. These findings underscore the complexity of the phase-ordering problem and suggest the necessity for ongoing research, potentially exploring deep learning models and more refined feature collection methods to further enhance compiler optimization strategies.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.
[2] https://link.springer.com/chapter/10.1007/978-3-030-55340-1_10
[3] https://dl.acm.org/doi/abs/10.1145/3124452
[4] https://github.com/EECS-583-Group-24/ML-LOOP