Facilitating Profile Guided Compiler Optimization with Machine Learning

Yang Yang
College of Computer Science and Technology, Jilin University
Changchun, Jilin, China

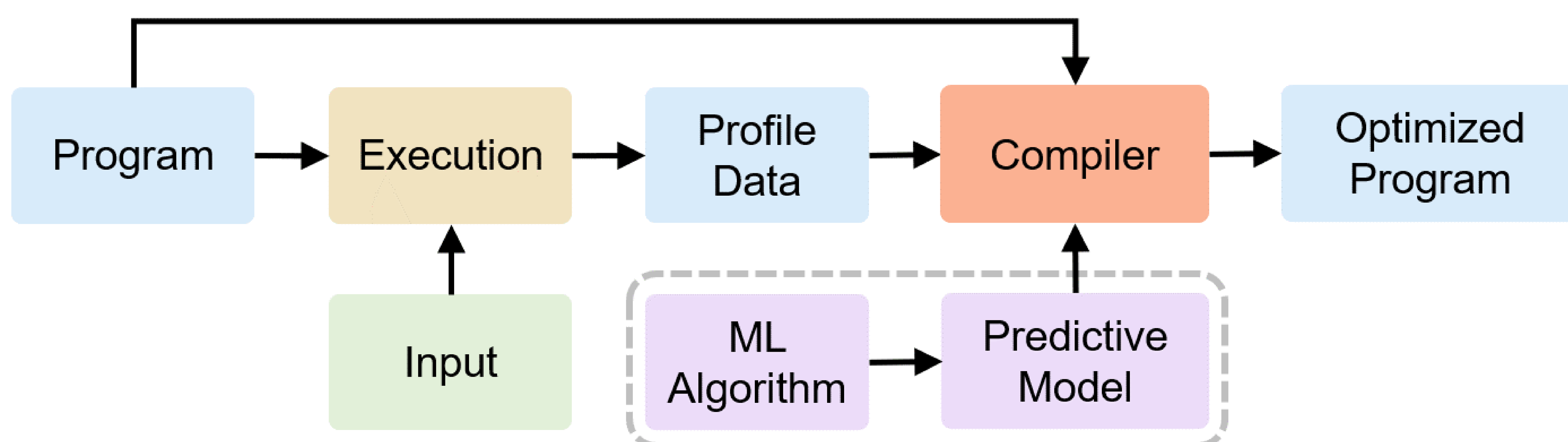
Xueying Wang
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Guangli Li*
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Introduction & Motivation

Profile guided optimization (**PGO**), which has proven to be effective in compilation for branch prediction, could improve program performance in most situations. However, traditional PGO methods need to collect numerous program runtime information through dynamic profiling for further compiler optimization. The cost of such a process is burdensome, it's usually fine-grained but time-costly, which is unacceptable for large programs. Therefore lightening such cost of PGO is meaningful.

In recent years, machine learning (ML) models have been introduced to guide compiler optimization, which can predict program information with slight cost. While delivering promising performance, there are numerous adjustable configurations when training ML models and integrating them with compilers, including model structures, features, and predicted categories. As such, it is still challenging to design an effective ML-aided compiler optimization system.



Before using ML to improve PGO, a few questions should be answered:

- ***Question1***: How to represent branch prediction task?
- ***Question2***: How to collect training data?
- ***Question3***: How to interact with the model?

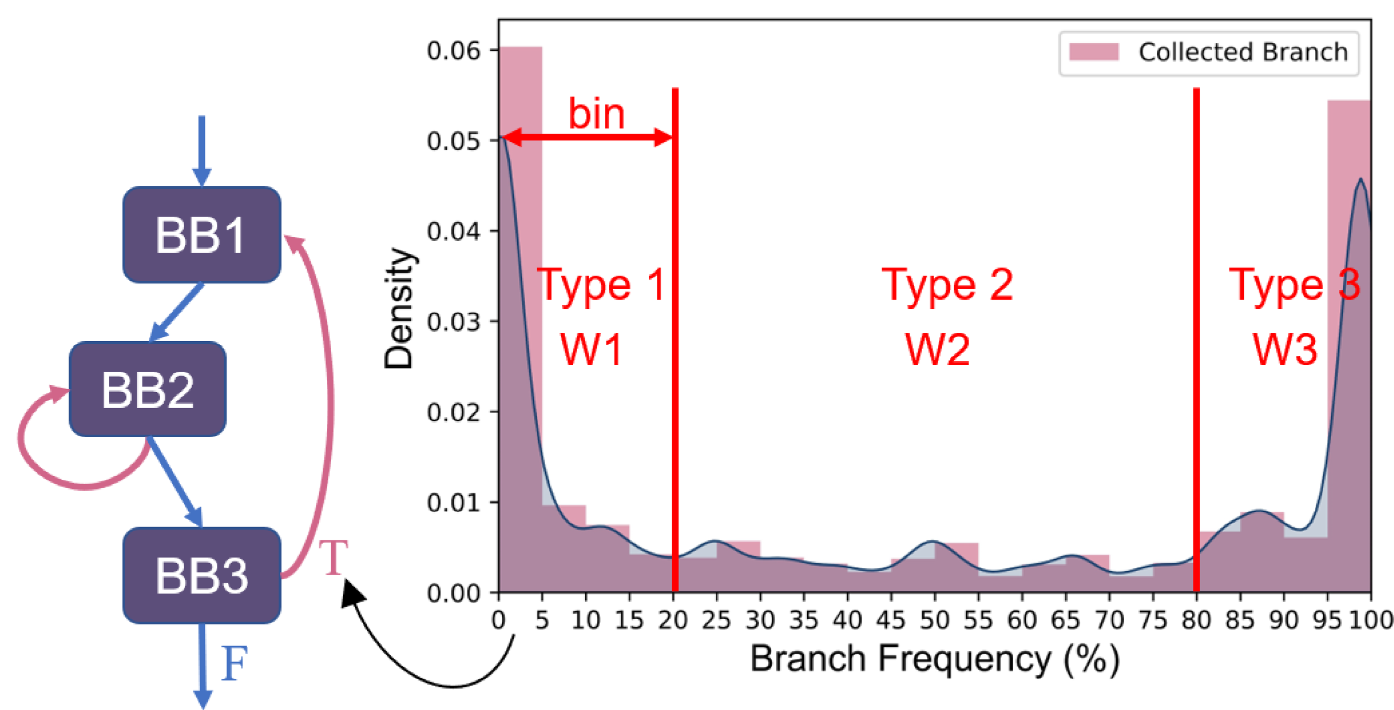
After determining the model and data used for our task, it is meaningful to explore the dynamic characteristics between branch prediction itself and the model's configuration. Therefore, a few more questions should be answered:

- ***Question4***: What is the label?
- ***Question5***: What is the criteria of our task?
- ***Question6***: What is the appropriate data format?

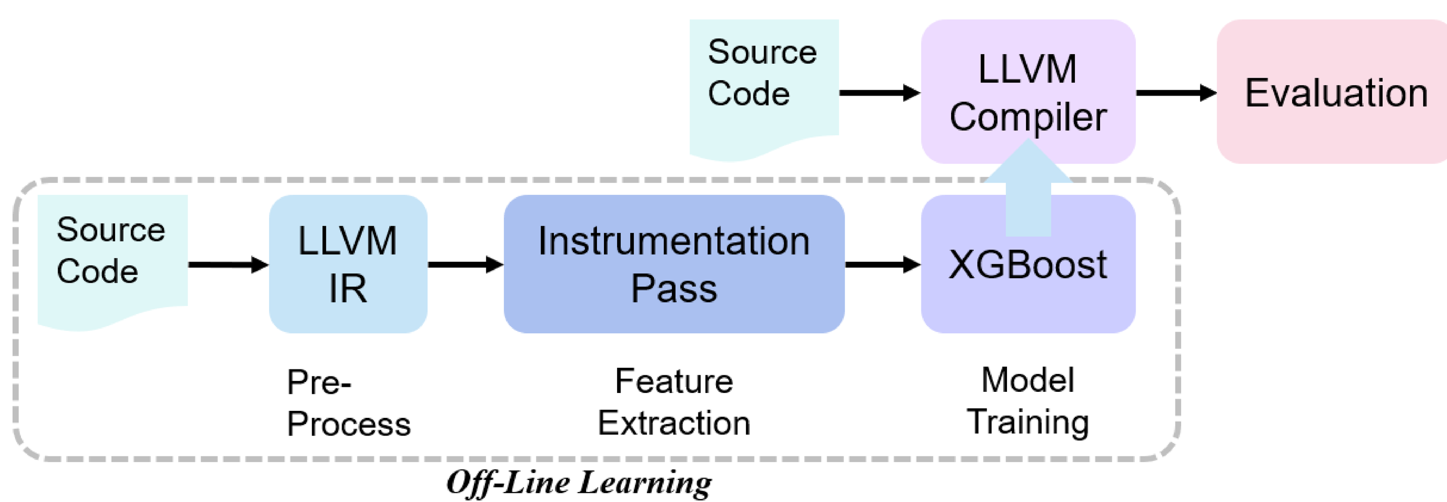
To answer these questions in facilitating PGO with machine learning using program's *static* properties, this poster presents an analysis to those questions.

Methodology

Answer 1: We have collected over 2,000,000 branches from existing benchmarks. From the distribution, we treat branch taken frequency as probability, divide it into several bins and transform the prediction task into a classification problem by assigning pre-defined branch weights.



Answer 2&3: We implement an instrumentation pass in LLVM, the information will be acquired when compiling which largely lightens the overhead of PGO. We choose XGBoost, a commonly-used ML model to build our predictive model, because it can provide a good balance between accuracy and overhead, and can be transferred to trees in C language, which is easy to integrate into LLVM source code.



Our Exploration

Answer 4&5: Based on ***Answer 1***, the branch density indicates a strongly two-head distribution, which guides us to form a three-way classification task. We transfer our task to predict the frequency and corresponding category. And we conduct an empirical analysis on the bin partition rules. Both intuition-based and equal division are tested, which proves that the three-way partition matches the distribution and performs better. And we find two representative programs which show 2 different kinds of performance variation (improve/decrease) when the partition changed.

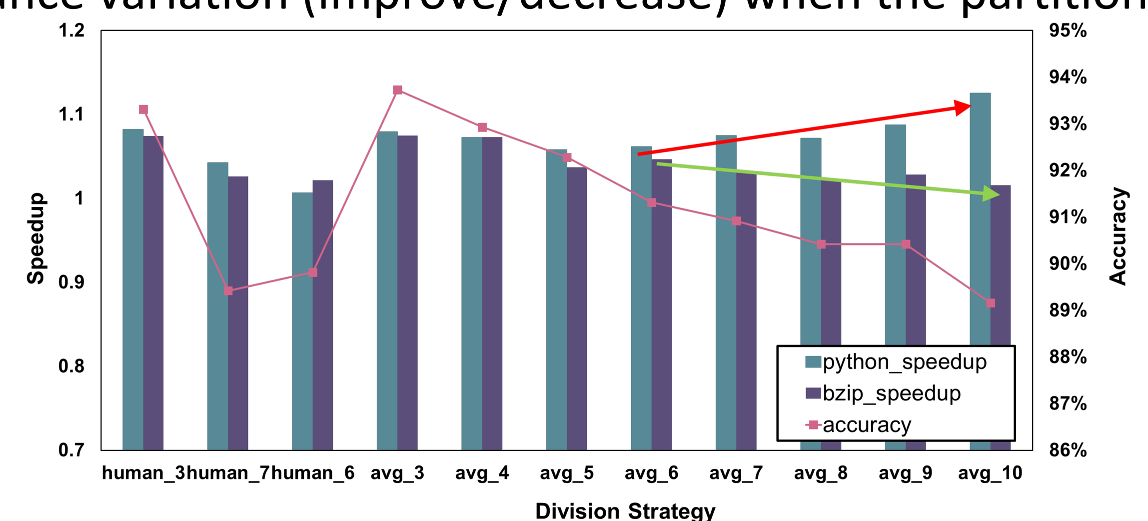


Fig. 1 Speedup with Different Division Strategies

Answer 6: We evaluate our model's sensitivity to the features and its format. By constructing a matrix of feature pair's Pearson product-moment correlation coefficients, 16 features are removed with only 1% accuracy decrease. And we keep removing features from training data iteratively to observe accuracy and speedup, the experimental results demonstrate that model's sensitivity to features is strongly program-dependent cause branch behavior differs from each other.

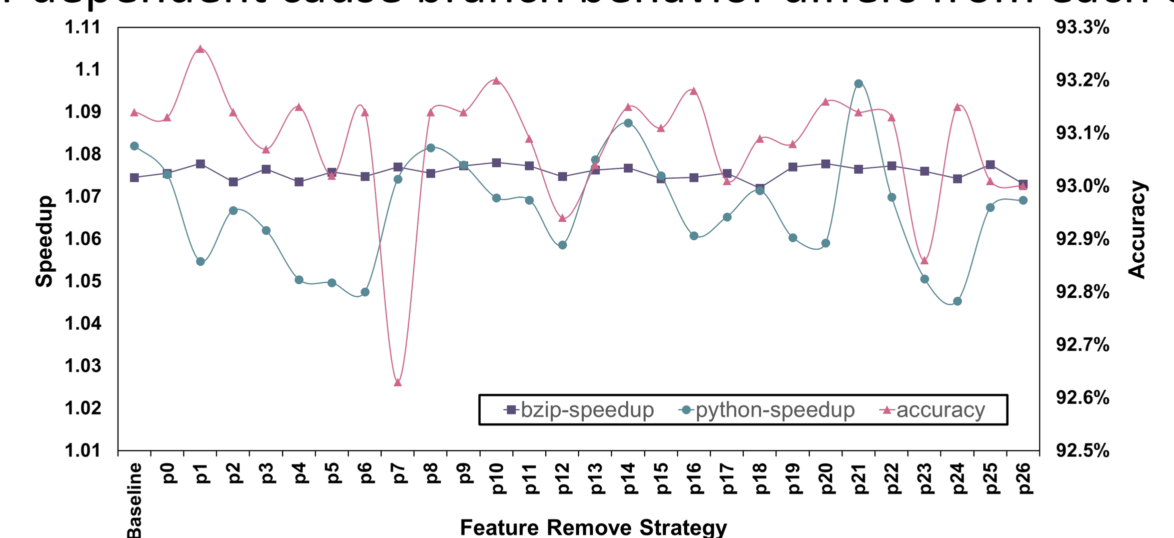


Fig. 2 Speedup with Different Feature Remove Strategies

Results & Conclusion

We implement a prototype system of ML-aided PGO based on the above analysis, which employs predicted weights, rather than realistic profiling weights, for branch probability. Evaluation with representative real-world applications and *Polybench* benchmark demonstrates the effectiveness of our method, achieving 1.03× and 1.95× speedup over the baseline (i.e., the programs without PGO), respectively. Moreover, the performance of our ML-aided PGO is very close to the classic PGO (1.05× and 1.97× speedups over the baseline) while reducing 58.3% and 94.8% optimization costs.

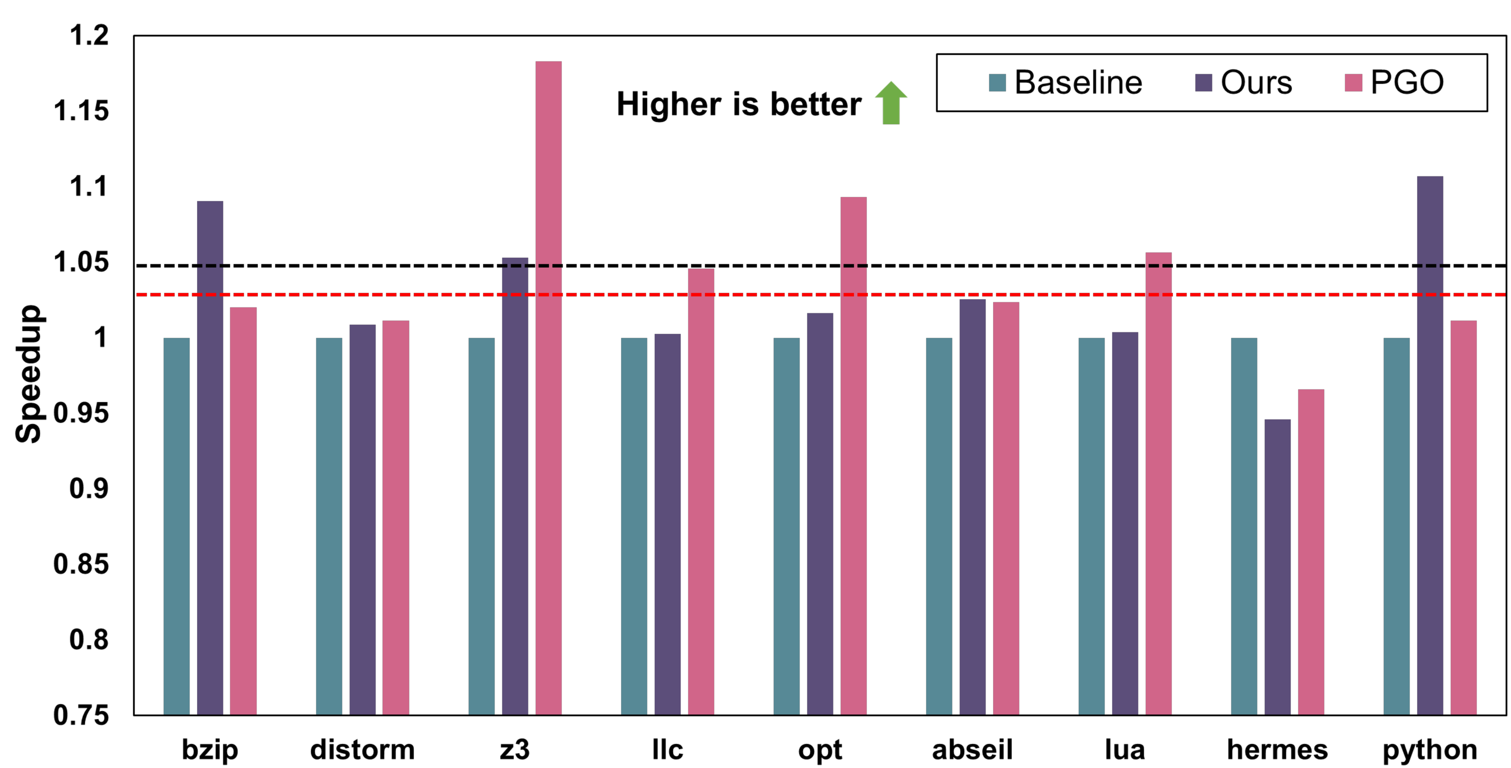


Fig. 3 Speedup on Real-World Applications

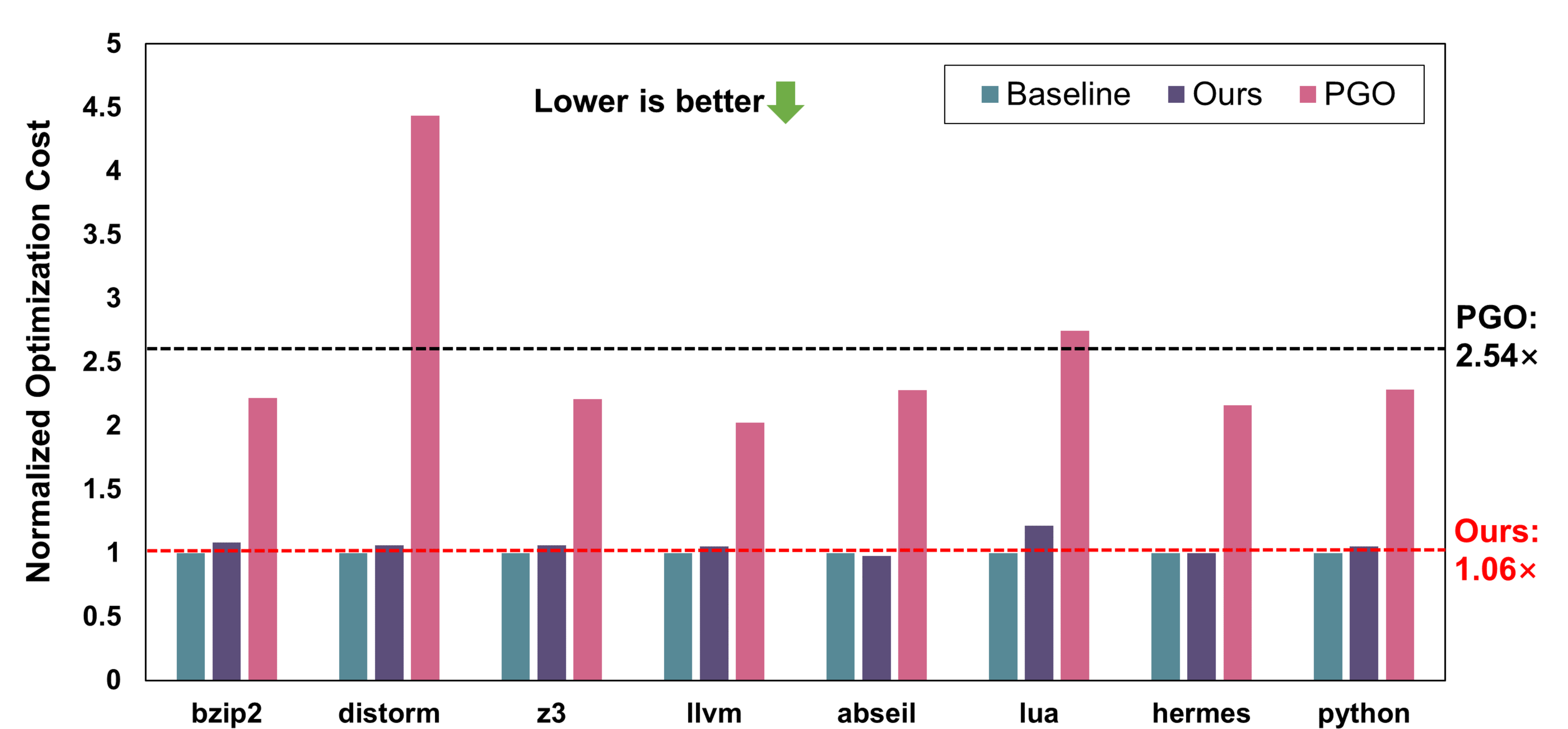


Fig.4 Optimization Cost on Real-World Applications