Profile Guided Optimization without Profiles: A Machine Learning Approach

By Nadav Rotem (Meta, Inc.) Chris Cummins (Meta AI)

Group5: Shuyuan Yang, Qifei Wu, Yijia Gao

Background: Problems for Classic PGO

Representative Profiling Data

Data collected during profiling does not accurately reflect real-world case

Overhead of Profiling

Require extra time/resources run program under different scenarios

Maintaining and Updating Profile Data

Re-profiling needed when new features introduced or bug fixed

Paper Overview

Novel statistical approach with Decision Tree:

- Infer branch probabilities => Decide which branch is hot
- Profiling => Supervised learning

Major steps:

- Step1: Offline training
  - Using branching information & Features collected from corpus of binaries
- Step2: Compiler use the trained model to estimate branching frequencies

Quick Overview for Decision Tree Model

Root Node:

Entire dataset

Splitting:

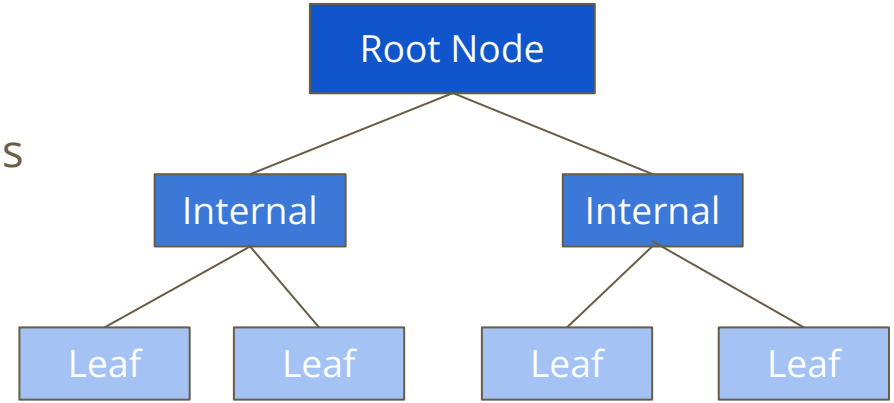
Dividing a node into more sub-nodes based on certain conditions

Internal Node:

After splitting, the sub-node that splits into further sub-nodes

Leaf/Terminal Node:

Nodes that do not split further, representing the decision outcome



Gradient Boosted Decision Tree

Advantages:

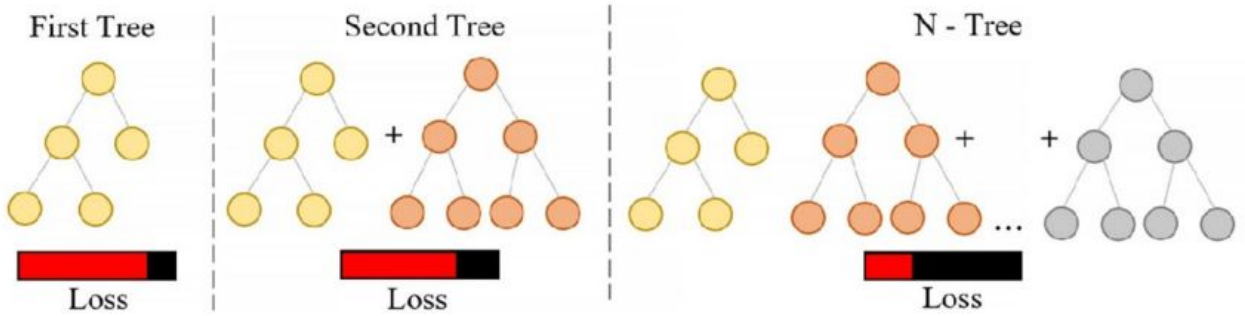
Resistant to overfitting, especially with large datasets

Provide more accuracy while keeping good run-time performance

Easier to integrate compared with CNN

Disadvantages:

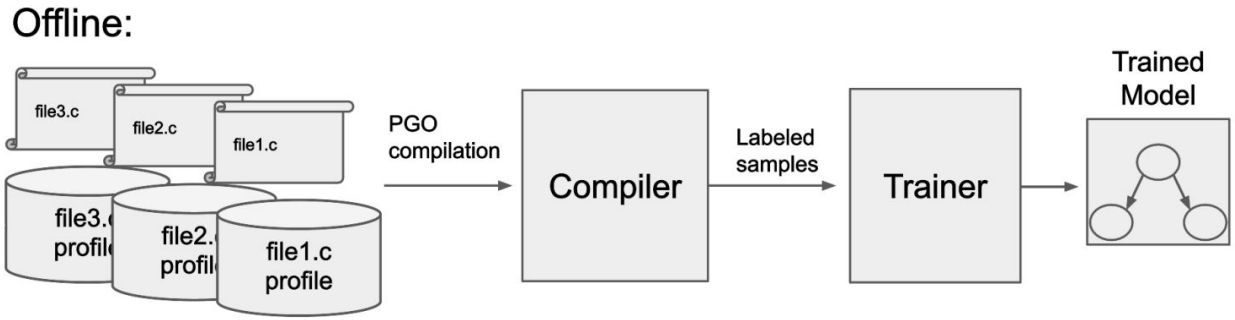
Less interpretable



Method - Two Phase Approach

Phase I: Offline training (Slow)

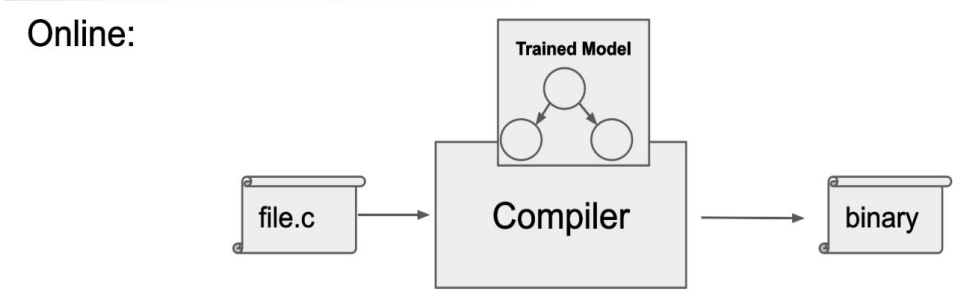
1. Collect high quality branch frequency data by PGO.
2. Custom feature collection pass to extract features.
3. Train a gradient boosted decision tree.
   a. Input: Extracted features related to branch.
   b. Output: Predict branch frequency.



Method - Two Phase Approach

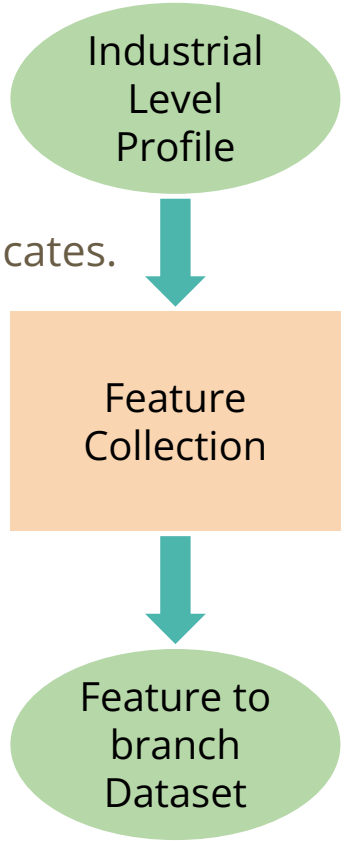
Phase II: Online compilation (Fast)

1. Regular compilation without PGO.
2. Collect features with the feature collection pass from input code file.
3. Use gradient boosted decision tree to predict branch frequency.



Feature Collection

- **Manually selected 34 features:**
  - 23 branch features: About the branch command itself.
  - 11 basic block feature: About the basic block the branch locates.
- **Wrote an analysis pass to collect all features:**
  - Represent each feature as a floating point number.
  - Collect all numbers into a “Feature Vector”.
- **Analyze a wide range of programs:**
  - High quality programs that are widely used
    - bzip2, Python, clang, sqlite, and more
  - Profile information also collected
  - Saved as training dataset



Train the Model

- Dataset with n examples and m features

$$D = \{x_i, y_i\}\ (|D| = n, x_i \in \mathbb{R}^m, y_i \in \mathbb{R}).$$

- Each decision tree considered as a function f: $f : \mathbb{R}^m \to \mathbb{R}$
- Split training set vs. testing set = 10 : 1
- Trained with XGBoost framework

Code Generation - From Tree to Code

Each decision tree is represented by 4 vectors:

- F: The feature to split on for each node
- C: The splitting condition. For leaf nodes it store the output of the tree.
- L: The left child of each node.
- R: The right child of each node.

The output of the gradient boost tree is the sum of output of all trees in it.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(X_i)$$

Simple loop with branches, very efficient!

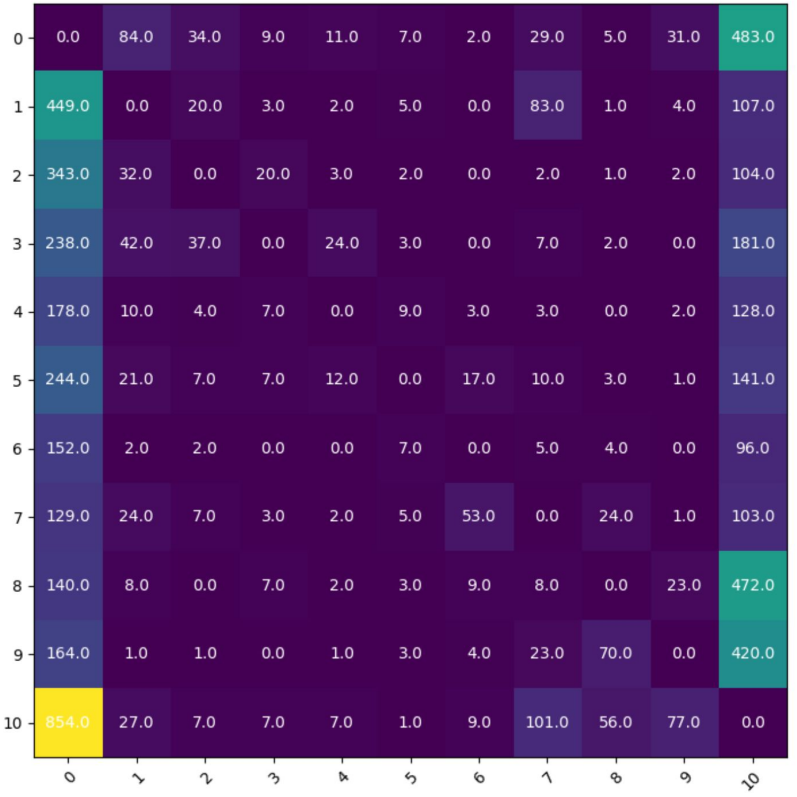
```
float intrp(const float *input,
            const short *F, const float *C,
            const short *L, const short *R) {
    int idx = 0;
    while (1) {
      if (F[idx] == -1) return C[idx]; // Found it!
      if (input[F[idx]] < C[idx]) // Check condition.
        idx = L[idx]; // Go left.
      else
        idx = R[idx]; // Go right.
    }
}
// Feature index, condition, left, right:
const short F0[] = {12, 26, 34, 28, 47, 18, 45,  ...
const float C0[] = {0.5, 0.5, 7.5, 1.5, 0.5, 0.5, ...
const short L0[] = {1, 3, 5, 7, 9, 11, 13, 15,17, ...
const short R0[] = {2, 4, 6, 8, 10, 12, 14, 16,   ...

float tree0(const float *input) {
 return intrp(input, F0, C0, L0, R0);
}
```

Evaluation - Two Experiments

- Predictive performance of branch weight model
- Run-time performance of programs compiled using predicted branch weights by the model

Evaluation - Predictive Performance

- Distribution of branch weight prediction errors
  - X-axis is ground truth class
  - Y-axis is predicted class
  - Brighter cells have higher density



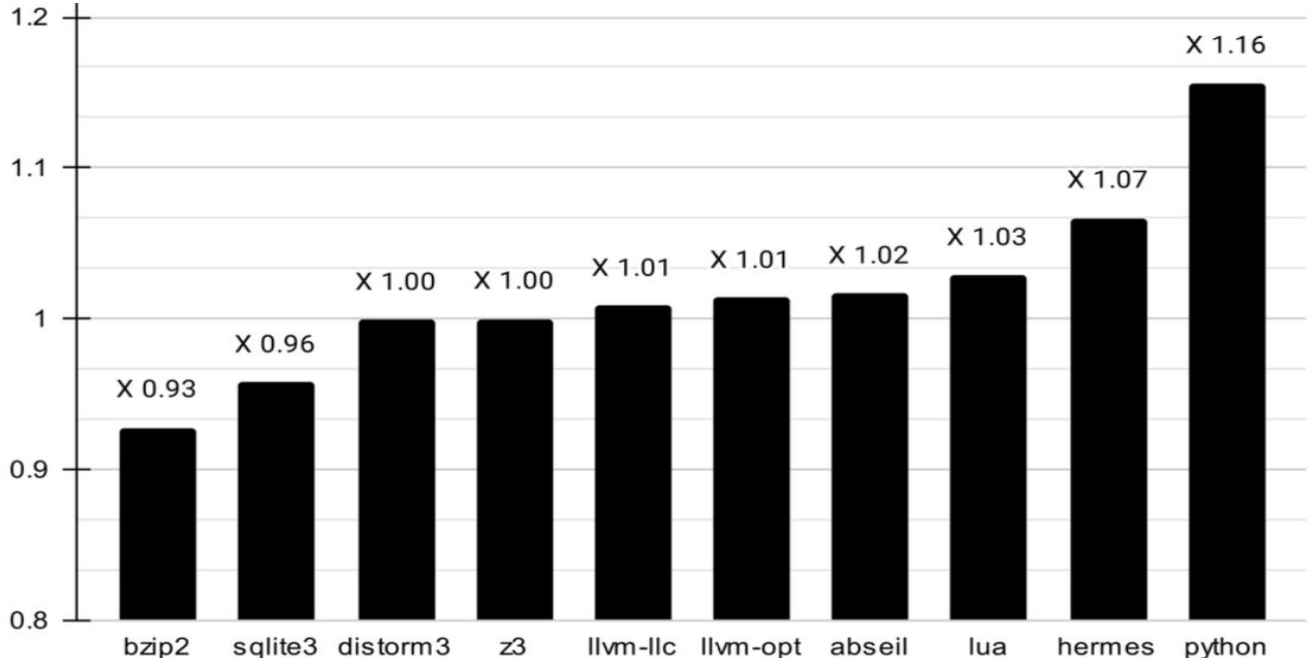
Evaluation - Predictive Performance

- Accuracy of the heuristics in LLVM's BPI analysis
  - ~ 10 million branches
  - Compare the outcome on the first successor for each branch

Heuristic	Branches	Weight	Correct
Estimated	2,965,889	0.29	0.70
Pointer	4,961,307	0.49	0.56
Zero	2,133,775	0.21	0.76
FloatingPoint	6,227	0.00	0.45

Evaluation - Run-time Performance

- Speedup of programs compiled using the proposed model over compilation without profile guided optimization.
  - The geometric mean speedup of our approach is 1.016.



Comparison

Challenge	This Research	Other Related Work
Data Extraction and Labeling	• Enables every measurement of a program to be used to produce ground truth labels for branch probability	• Relies on exhaustive compiling and measuring • Determines the best performance on every combination of optimization decision
Dealing with Measurement Noise	• Focuses on noise-free deterministic branch probabilities ground truth	• Measurement noise as a prediction target • Relies on proxy metrics like static analysis of the generated binaries

Comparison

Challenge	This Research	Other Related Work
Feature Design and Selection	• Pairs abundant values from LLVM's static analysis with gradient boosted trees to balance interpretability and featuring cost • Automatically ranks and prunes various features	• Handcrafts vectors of numeric features • Sacrifices interpretability for simplification by inferring high-level features from low-level representations with natural language models or graph learning

Conclusion

The approach developed in this research ...

- Leverages gradient-boosted trees for efficient branch prediction.
- Has low compile times and zero additional memory overhead.
- Is easy to train and integrate into compilers.
- Can be further explored on tradeoffs in each stage.

Appendix - List of All Features

Type	List of features
Branch Features	is_entry_block
	num_blocks_in_fn
	condition_cmp
	condition_predicate
	condition_in_block
	predicate_is_eq
	predicate_is_fp
	cmp_to_const
	left_self_edge
	right_self_edge
	left_is_backedge
	right_is_backedge

Type	List of features
Branch Features	right_points_to_left
	left_points_to_right
	loop_depth
	is_loop_header
	is_left_exiting
	is_right_exiting
	dominates_left
	dominates_right
	dominated_by_left
	dominated_by_right
	num_blocks_dominated

Type	List of features
Basic Block Features	num_instr
	num_phis
	num_calls
	num_loads
	num_stores
	num_preds
	num_succ
	ends_with_unreachable
	ends_with_return
	ends_with_cond_branch
	ends_with_branch
	num_instr

Citation

[1] Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In ICML, 2021.

[2] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end Deep Learning of Optimization Heuristics. In PACT, 2017.

[3] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In ICML, 2019.

[4] Rotem, N., & Cummins, C. (2021). Profile Guided Optimization without Profiles: A Machine Learning Approach. ArXiv. /abs/2112.14679

Thank you