Static Branch Prediction for LLVM IRs Using Machine Learning

Ching-Yen Shih cyshih@umich.edu
Drake Svoboda drakes@umich.edu
Siao-Jie Su siaojie@umich.edu
Wei-Chung Liao wchliao@umich.edu

Group 12

Abstract

Branch prediction is crucial in many optimizations, and evidence-based branch prediction methods have been demonstrated to be superior to program-based heuristic methods. Evidence-based methods predict branch behavior using machine learning techniques. Calder et al. introduce evidence-based prediction [1]; however, they only use a limited set of features and machine learning models.

In this paper, we apply a broader feature set and a diverse set of machine learning models for evidence-based prediction. We show that extending the number of features and using more powerful machine learning models can improve performance. We also found that graph neural networks (GNN) can achieve comparable performance to some classical ML models even without predefined features.

1 Introduction

Static branch prediction is the process of predicting whether branches will be taken or not before they are actually executed. There are many optimizations that rely on branch prediction and execution estimation, such as trace selection [2] and superblock formation [3, 4]. Without a doubt, branch prediction is important.

Typically, compilers rely on two general approaches for estimating branch behavior at compile time: profile-based and program-based branch prediction. Profile-based methods use program profile information to determine the frequency that branch paths are executed. Profile-based branch prediction is accurate at predicting the future behavior of branches. However, profile-based methods require additional work to generate the program profiles. Program-based methods, on the other hand, attempt to predict the behavior of branches using heuristics based on only the structure of a program. Thus, unlike profile-based methods, program-based methods do not need profile information.

Calder et al. [1] proposed evidence-based static branch prediction, a new program-based branch prediction method, which predicts branch behavior using machine learning. Evidence-based branch prediction outperforms all existing program-based methods at the time. In addition, evidence-based branch prediction is very general and can be applied to a wide range of program behavior estimation problems. However, Calder et al. [1] only used few features and machine learning models in their paper.

In this final project, we furthermore apply a broader feature set and a diverse set of machine learning models, including some state-of-the-art deep learning models, for evidence-based prediction. Our experiment results show that the machine learning models outperform LLVM's heuristic method by a near 40% in miss rate. Also, each of our models outperforms decision tree model, which is the model used by Calder et al. [1]. Finally, we also show that graph neural network (GNN) could achieve comparable performance to some classical machine learning models even without any predefined features.

This report has the following organization. In Section 2 we discuss some background and related work of different kinds of static branch prediction methods. The motivation of our final project is discussed in Section 3, and we discuss the methods we use in Section 4. Section 5 elaborates the settings in our experiments, and Section 6 presents our evaluation and results. Finally, we summarize our conclusions in Section 7.

2 Background & Related Work

In this section, we discuss some existing approaches to profile-based static branch prediction, program-based static branch prediction and evidence-based static branch prediction. Machine learning is also discussed with evidence-based static branch prediction in this section.

2.1 Profile-based Branch Prediction Methods

Profile-based methods predict branch behavior based on program profiles. One of the simplest and typical profile-based methods is to measure the frequency of each path, and for each branch we predict the branch behavior the same as the profile data indicates — if the frequency of the branch being taken is more than the frequency of it being not taken, we predict the branch to be taken. We predict the reverse otherwise.

Profile-based branch prediction is shown to be extremely successful at predicting branch behaviors. However, profile-based methods rely on program profiles, which requires additional work to generate. It is usually time-consuming and costly.

2.2 Program-based Branch Prediction Methods

Program-based methods attempt to predict branch behavior using heuristics based on only the structure of a program. Thus, unlike profile-based methods, program-based methods do not need profile information.

Ball and Larus [5] used a number of simple program-based heuristics to predict the behavior of branches. While simple, the predictor is quite successful.

Nevertheless, two questions come with such program-based approaches. The first question is which heuristics should be used. Ball and Larus proposed seven heuristics that they considered successful, but they also noted that they tried many heuristics that were not successful. [5] The second question is how to decide what to do when more than one heuristic can be applied to a given branch. In machine learning this is known as the "evidence combination" problem. Ball and Larus decided that the heuristics should be applied in a fixed order, but apparently it is not guaranteed to be the best order.

In addition, Calder et al. pointed out that such program-based heuristics approaches are sensitive to differences in architecture, compiler, run-time system, and selection of programs [1]. In other words, the heuristics may fail when architecture, compiler, run-time systems or selection of programs changes. This sensitivity motivates us to develop techniques that can automatically select heuristics for branch prediction.

2.3 Evidence-based Branch Prediction Methods

Evidence-based methods [1] predict the behavior of new programs according to the corpus of programs. It solves the questions and problems of program-based methods mentioned above — it can choose and combine heuristics (features) automatically. Therefore, programmers do not need to worry about which heuristics should be used and what to do when more than one heuristics is applicable to a branch. Also, since it is done automatically, it is simple to apply this method on different architectures, compilers, and programming languages. For example, if we want to do the branch prediction on the new compiler, we can directly collect the corpus of training data and then train the model rather than find out the heuristics manually.

On top of it, evidence-based branch prediction is very general and can be applied to a wide range of program behavior estimation problems. These advantages prove the usefulness of evidence-based methods.

There are two important parts in the evidence-based method, namely, feature extraction and machine learning model. We can extract important information from the original programs through feature extraction, and then predict the branch by feeding those features into the machine learning model. For instance, Calder et al. [1] achieve a miss rate of 20% by using decision tree as the model and found that the opcode of the terminating instruction in the successor is the most important features in C programs. Desmet et al. [6] get miss rate of 18.5% by decision tree as well.

Recently, deep learning achieves state-of-the-art performance on many tasks, such as natural language processing and computer vision. One of the advantages of deep learning is that the model doesn't require manual feature extraction and can automatically learn feature representations from the corpus of data [7]. Hence, there are some research on branch prediction by deep learning. For example, Mao et al. [8] employs both deep convolutional neural networks (CNNs) and deep belief network (DBN) for branch prediction.

3 Motivation

The evidence-based method proposed by Calder et al. [1] was successful. However, they only used few features and machine learning models in their paper. Although they demonstrated decent results, we believe that we can improve the performance by extending the number of features and using more machine learning models, including some state-of-the-art deep learning models.

In next section, we discuss the features and the machine learning models we use.

4 Method

We apply both classical machine learning methods and a deep-learning method to the static branch prediction problem. In this section we describe our models and how we extract features from an LLVM IR. We also compare deep-learning to classical machine learning methods and weigh the benefits of using deep neural networks.

4.1 Classical Machine Learning

Feature Extraction We have written an LLVM pass that produces a descriptive feature vector for each branch instruction in an LLVM IR. Our feature vector has a total of 52 features. In comparison, Calder et al. use a feature representation with only 30 features [1]. Each of our features are easily extracted using LLVM libraries. Table 3 shows the comprehensive list of our features. Our features can be grouped into five categories: floating point comparison, integer comparison, pointer comparison, loop, and successor features. Each successor feature is included for both the taken and fall-through successors of a branch.

Many of our features are derived from existing heuristical methods. Our pointer comparison features and loop features follow directly from the Ball and Larus heuristic [5]. We also derive features from LLVM's internal branch prediction methods. For example, one heuristic LLVM uses is that certain call instructions are designated as "cold" or unlikely to be executed. Thus, LLVM internally predicts that basic blocks containing these calls are unlikely. We adapt this heuristic into the binary feature `succ_post_dom_by_cold_call` which is true when the branch's successor either contains a cold call or is post dominated by a basic block with a cold call. Another heuristic LLVM uses is that certain library functions like `memcmp` and `strcmp` are unlikely to return true. We similarly adapt this heuristic to the features `is_icmp_eq_lib` and `is_icmp_ne_lib`.

Models We apply 5 classical machine learning models to predict the branch, namely, decision tree [9], K-nearest neighbors [10], random forest [11], support vector machine [12, 13], and XGBoost [14].

Decision trees [9] generate a tree for prediction where each feature maps to a node in the tree. The leaves in the decision tree represent the prediction result. The decision tree makes predictions based on a serial process. Therefore, it is interpretable more than other models, and can be easily visualized. This can be directly compared to a sequential ordering of heuristics; however, the decision tree is able to learn many orderings via multiple paths through the tree.

K-nearest neighbors (KNN) [10] is a non-parametric method. It predicts the result by observing the neighbors of the input data. In KNN classification, the input data is classified by a plurality vote of its nearest K neighbors.

Random forest [11] is the ensemble method. There are lots of decision trees in the forest, and the output is decided by majority vote of each decision tree. There are some randomness in each decision tree to increase the diversity of the forest. For example, the training data and features in each decision tree are different, so the trees are diverse.

Support vector machines(SVM) [12, 13] map the data as points in space then construct a hyperplane that separates two classes as wide as possible for the classification problems.

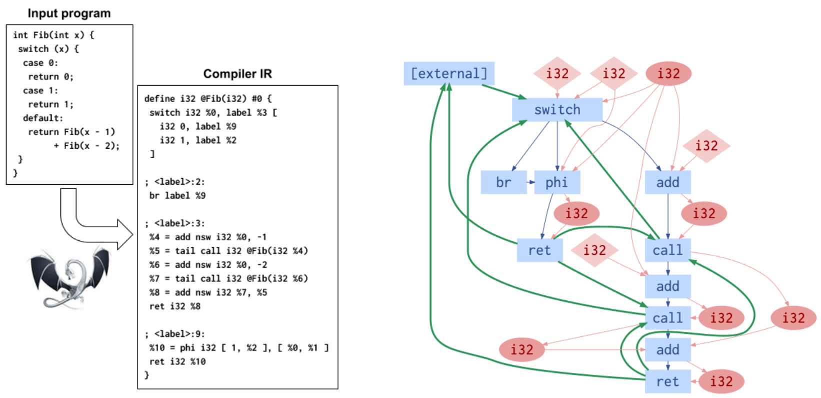
XGBoost [14] is an efficient and scalable gradient boosting machine. It is also an ensemble method consisting of classification and regression trees (CART). Therefore, like decision trees, it can also be visualized.

4.2 Deep Learning

Classical machine learning approaches resolve many problems with heuristic based approaches for branch prediction; however, classical ML methods require us to manually define a feature representation for a branch instruction. Determining the best features representation still requires heuristics and expert understanding of programming languages and architectures. We try to resolve this problem by leveraging deep-learning. One major benefit of deep-learning over classical machine learning is that deep-learning models can automatically learn feature representations without the need

Feature Name	Description
Floating Point Comparison Features	
is_fcmp	True if the condition is a floating point comparison
is_fcmp_eq	True if floating point equality
is_fcmp_ne	True if floating point inequality
is_fcmp_nan	True if floating point equality with nan
is_fcmp_not_nan	True if floating point inequality with nan
Integer Comparison Features	
is_icmp	True if the condition is integer comparison
is_icmp_cnst	True if integer comparison with a constant
is_icmp_cnst_one	True if comparison with one
is_icmp_lt_one	True if less than comparison with one
is_icmp_cnst_zero	True if comparison with zero
is_icmp_eq_zero	True if equality comparison with zero
is_icmp_ne_zero	True if inequality comparison with zero
is_icmp_gt_zero	True if greater than comparison with zero
is_icmp_lt_zero	True if less than comparison with zero
is_icmp_cnst_minus_one	True if comparison with minus one
is_icmp_eq_minus_one	True if equality comparison with minus one
is_icmp_ne_minus_one	True if inequality comparison with minus one
is_icmp_gt_minus_one	True if greater than comparison with minus one
is_icmp_lib	True if comparison with the return code of `strcmp`, `memcmp`, `strncmp`, `strcasecompare`, or `strncasecmp`
is_icmp_eq_lib	True if equality with the return code of an above library function
is_icmp_ne_lib	True if inequality with the return code of an above library function
Pointer Comparison Features	
is_pointer_compare	True if pointer comparison
is_pointer_compare_eq	True if pointer equality comparison
is_pointer_compare_ne	True if pointers inequality comparison
Loop Features	
is_block_in_loop	True if the branch instruction belongs to a loop
loop_back_edge	True if the edge to the successor is a loop back edge
loop_entering_edge	True if the edge to the successor enters a loop
loop_exiting_edge	True if the edge to the successor exits a loop
succ_exits_loop	True if the successor starts unconditional control flow that exits a loop
succ_loop_back_edge	True if the successor starts unconditional control flow that takes a back edge
Successor Features	
succ_post_dom_by_cold_call	True if the successor is post-dominated by "cold" function call
succ_post_dom_by_unreachable	True if the successor is post-dominated by "unreachable" instruction
succ_alloca	True if the successor contains an alloca instruction
succ_call	True if the successor contains a call instruction
succ_return	True if the successor contains a return
succ_store	True if the successor contains a store
succ_dominates	True if the successor dominates the basic block
succ_post_dominates	True if the successor post-dominates the basic block
succ_ends	The type of instruction that terminates the successor
succ_use_def	True if the successor contains a use for a value defined in the basic block

Table 1: Static feature set. Each successor feature is included for both the taken and fall-through successor.



```
int Fib(int x) {
 switch (x) {
  case 0:
   return 0;
  case 1:
   return 1;
  default:
   return Fib(x - 1)
        + Fib(x - 2);
 }
}
```

```
define i32 @Fib(i32) #0 {
 switch i32 %0, label %3 [
   i32 0, label %9
   i32 1, label %2
 ]

; <label>:2:
 br label %9

; <label>:3:
 %4 = add nsw i32 %0, -1
 %5 = tail call i32 @Fib(i32 %4)
 %6 = add nsw i32 %0, -2
 %7 = tail call i32 @Fib(i32 %6)
 %8 = add nsw i32 %7, %5
 ret i32 %8

; <label>:9:
 %10 = phi i32 [ 1, %2 ], [ %0, %1 ]
 ret i32 %10
}
```

Figure 1: ProGraML [15] representation for a sample IR. First the program is compiled to an LLVM intermediate representation. To construct the graph, each instruction, identifier, and value is added as a vertex the vertices are connected via control-flow, data-flow and call-flow edges.

for manual feature extraction [7]. This is one reason why deep-learning offers state-of-the-art performance for many natural language processing and computer vision tasks [7].

We propose using a graph neural network (GNN) for predicting branch bias on a graph-based representation of LLVM IRs. We choose ProGraML [15] as our graph-based representation. ProGraML represents an IR as a directed graph where each instruction, identifier, and value is a vertex and each vertex is connected via control-flow, data-flow and call-flow edges. Figure 1 shows a ProGraML representation of an example program. To pass through our model, each vertex is embedded to learnable embedding with 32 dimensions using an augmented inst2vec [16] vocabulary. The entire graph is then passed through 8 generalized graph convolutional layers defined by Li et al. [17]. Each layer sequentially updates each vertex's hidden feature representation by aggregating information from adjacent vertices. Then, for each vertex that corresponds to a branch instruction, we pass its final feature representation through a linear classifier that predicts the branch's bias. The entire model has 107,914 trainable parameters.

5 Experiments

Dataset For training and evaluating our models we collect a corpus of 21 programs written in the C programming language. The programs in our corpus were taken from the cBench and Paribol [18] benchmark suites. The programs are compiled with profiling instrumentation using the Clang compiler. We compile using clang's `-emit-llvm` flag to generate LLVM IRs. This yields a corpus with a total of 2,589 conditional branch instructions. To generate ground truth labels for each branch instruction, we merge profiling data from multiple runs using program inputs supplied by the benchmark suites. To generate graph representations for the GNN, we use the command line tools supplied in ProGraML's [15] public GitHub repository. Table A1 shows some more basic statistics for our dataset.

Classical Machine Learning Models We trained five classical machine learning models for branch prediction: decision tree, SVM, random forest, XGBoost, KNN. We used the scikit-learn python package to train these models. We choose hyper-parameters by conducting leave-one-out cross validation.

Graph Neural Network The graph neural network is trained for 2,000 iterations using the AdamW optimizer [19]. At each iteration, a training batch is constructed by randomly selecting a single program from the corpus. The ground truth label for each conditional branch is a binary random variable for left-biased or right-biased. The loss is computed as the average cross-entropy over each conditional branch in the program. Labels are extracted from profiling data. An L2 penalty is applied during training with $\lambda = 0.01$. The learning rate is fixed at $1e - 3$.

6 Evaluation & Results

To evaluate our models we conduct leave-one-out cross validation; that is, we evaluate the performance on each program in our corpus by training on all other programs. Table 4 shows our cross validation results. The XGBoost classifier achieves the best performance among all of the models as shown in table 4. The performance of random forest is

KNN	Decision Tree	Random Forest	SVM	XGBoost	LLVM heuristics	Perfect
35.58	38.79	33.12	37.15	**31.44**	54.66	6.28

Table 2: Macro. Average leave-one-out cross-validation miss rate (%) on hard-to-predict branches. A hard-to-predict branch is a branch that does not have a loop exiting edge or loop back-edge and is not post-dominated by an unreachable instruction. Each model considerably out-performs the baseline LLVM heuristic method.

Ranking	Feature name	Description
1	is_right_loop_exiting_branch	Does the right control-flow edge exit a loop
2	is_left_post_dom_by_unreachable	Is left succ. post-dominated by unreachable instruction
3	left_succ_ends_unreachable	Is the type of instruction that terminates the left successor unreachable

Table 3: Top 3 features ranked by XGBoost classifier

comparable to XGBoost. Both of these classifiers ensemble predictions from several small models; generalizing a single classifier is difficult, so ensemble-methods are helpful for reducing the generalization error of branch prediction. The single decision tree performed the worst among all of the models. Decision tree is the classifier used in *Evidence-based Static Branch Prediction using Machine Learning* [1]. Our results show that it is beneficial to explore methods beyond a single decision tree.

The best average miss rate of our method is 27.31%, which is much better than LLVM heuristics which achieves a miss rate of 44.04%. However, there is still a large space for improvement since a perfect miss rate is 7.42%.

The GNN's performance matched that of the classical methods. We believe that increasing the size of our corpus could considerably improve the GNN's performance. The GNN has many more parameters then the classical models and therefore requires more training data. Despite the underwhelming performance compared to the classical methods, we still believe that GNN-based approaches show promise for compiler prediction tasks.

Features importance Table 3 shows the top three most important features ranked by the XGBoost classifier. The most important feature is `is_right_loop_exiting_branch`. This makes sense intuitively since loop exiting branches are likely to be heavily biased. The second and third best features are `is_left_post_dom_by_unreachable` and `left_succ_ends_unreachable` respectively. Hence, our model will choose to take the right branch if left successor is post-dominated by an unreachable instruction or and unreachable instruction terminates the left successor. Again, this makes intuitive sense since unreachable instructions are guaranteed not to execute.

Hard-to-predict branches Heuristic methods are good at predicting "easy-to-predict" branches where a single powerful heuristic—like a loop exiting heuristic—can achieve good results. Machine learning can be very beneficial for "hard-to-predict" branches where a single heuristic is not enough and heuristic methods must rely on a combination of several less powerful heuristics. In this section we evaluate our model on these "hard-to-predict" branches. In the previous section we show that `is_right_loop_exiting_branch`, `is_left_post_dom_by_unreachable`, and `left_succ_ends_unreachable` are the most important features for the XGBoost classifier. Thus, we define a "hard-to-predict" branch as a branch that does not have loop exiting or loop back edges and whose successors are not post-dominated by unreachable instructions. Table 2 shows the results on these types of branches; the miss rate of each classifier degrade by about 4 to 6 percentage points. The LLVM heuristic method performs quite poorly in this case and misses more than 50% of branches. The XGBoost classifier still has the best performance among all of the classifiers with an average miss rate of 31.44%. This result shows a major benefit over heuristic based approaches.

7 Conclusion

Our experimental results show that ML methods considerably out-perform LLVM's heuristic method by a near 40% decrease in miss rate (16.73 percentage point decrease). Our models especially showed improvement on "hard-to-predict" branches where a single powerful heuristic fails to correctly predict bias. Ensemble-based models (random forest and XGBoost) perform best amongst our models; hence, ensemble-methods are helpful for reducing generalization error on branch prediction.

Also, Calder et al. [1] propose evidence-based static branch prediction; in their paper they use a limited and manually defined feature set and only try a neural network and decision tree. We use a more diverse feature set and selection of

Program	KNN	Decision Tree	Random Forest	SVM	XGBoost	GNN	LLVM Heuristics	Perfect
automotive_bitcount	26.15	25.60	25.60	**17.94**	26.15	41.69	27.47	3.92
automotive_qsort1	**21.34**	37.56	31.10	34.34	27.77	32.31	43.10	9.26
automotive_susan	**14.09**	70.89	11.85	73.54	15.58	71.15	70.31	6.72
bzip2d	28.68	26.30	27.38	**25.79**	29.55	37.32	42.55	7.80
consumer_jpeg	33.28	**30.27**	32.49	33.34	32.12	39.76	43.87	8.37
consumer_tiff	48.46	48.36	44.64	45.63	**30.36**	52.98	46.90	7.34
cutcp	18.65	17.49	**12.24**	13.49	13.49	31.23	45.50	7.33
histo	35.74	36.65	**33.46**	**33.46**	**33.46**	42.26	45.58	7.56
mri-gridding	22.16	29.24	25.37	25.37	**20.55**	43.67	45.47	7.53
network_dijkstra	**5.49**	22.10	11.04	**5.49**	11.04	36.99	44.97	7.45
network_patricia	30.23	36.38	**21.63**	24.22	24.84	52.68	44.71	7.54
office_stringsearch1	19.98	18.89	19.73	19.73	19.73	**16.38**	44.17	7.48
sad	20.21	19.44	**14.20**	18.97	18.97	23.95	43.22	7.63
security_blowfish	25.03	18.78	18.78	18.78	18.78	**14.87**	43.05	7.56
security_rijndael	22.33	32.79	14.45	14.45	**11.82**	30.60	42.66	7.44
security_sha	19.95	**14.07**	19.95	19.95	19.95	14.38	42.36	7.43
spmv	32.04	30.09	**27.16**	28.48	30.82	35.73	42.46	7.45
stencil	15.86	18.24	**11.10**	15.86	13.48	14.26	42.13	7.42
telecom_CRC32	**7.15**	**7.15**	**7.15**	**7.15**	21.43	**7.15**	41.85	7.15
telecom_adpcm	22.11	22.22	**16.13**	**16.13**	22.38	25.38	41.63	7.58
tpacf	16.46	18.17	**7.71**	11.22	16.48	20.67	40.93	7.52
Macro Avg.	30.75	32.31	28.45	31.20	**27.31**	32.64	44.04	7.42

Table 4: Leave-one-out cross validation miss-rates. For each program, we report the miss rate (%) achieved after training on the other programs. Miss rate is computed as the profile count of the not-predicted control flow direction over the total profile count of the branch. The "LLVM Heuristics" column shows the performance of LLVM's heuristic-based prediction methods. The "Perfect" column reports the miss-rates when using the profile data directly to predict branches.

machine learning models. Our results are not directly comparable; however, each of our models out-performs a decision tree baseline on our corpus. This suggests that using a more diverse set of models is beneficial for branch-prediction.

Our deep-learning approach had comparable performance to our classical-models. Deep neural networks require large amounts of training data due to their size. Although our graph-neural network did not show improvement over the classical methods, we believe that GNNs could learn better feature representation than manually defined features if we increase the size of our training corpus. Thus, deep-learning is a promising direction for compiler prediction tasks.

References

[1] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems*, vol. 19, 12 1996.

[2] P. P. Chang and W. W. Hwu, "Trace selection for compiling large c application programs to microcode," in *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, MICRO 21, (Washington, DC, USA), p. 21–29, IEEE Computer Society Press, 1988.

[3] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for vliw and superscalar compilation," *J. Supercomput.*, vol. 7, p. 229–248, May 1993.

[4] R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhaal, and W.-m. Hwu, "Superblock formation using static program analysis," pp. 247 – 255, 01 1994.

[5] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, (New York, NY, USA), p. 300–313, Association for Computing Machinery, 1993.

[6] V. Desmet, L. Eeckhout, and K. De Bosschere, "Using decision trees to improve program-based and profile-based static branch prediction," in *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pp. 336–352, Springer, 2005.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, 05 2015.

[8] Y. Mao, H. Zhou, X. Gui, and J. Shen, "Exploring convolution neural network for branch prediction," *IEEE Access*, vol. 8, pp. 152008–152016, 2020.

[9] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, p. 81–106, Mar. 1986.

[10] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[11] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, p. 5–32, Oct. 2001.

[12] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, p. 273–297, Sept. 1995.

[13] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[14] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), p. 785–794, Association for Computing Machinery, 2016.

[15] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," 2020.

[16] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," 2018.

[17] G. Li, C. Xiong, A. Thabet, and B. Ghanem, "Deepergcn: All you need to train deeper gcns," 2020.

[18] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, vLi Wen Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Impact technical report," 2012. Benchmarks available at `http://impact.crhc.illinois.edu/parboil/parboil.aspx`.

[19] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

A Dataset

Program Name	# of Instructions	# of Conditional Branches	% of Conditional Branches	# of Data Sets
cBench Benchmark Suit				
automotive_bitcount	756	12	1.59	1
automotive_qsort1	534	31	5.81	20
automotive_susan	12685	107	0.84	20
bzip2	26508	837	3.16	1
consumer_jpeg	55028	500	0.91	40
consumer_tiff	53515	433	0.81	90
network_dijkstra	410	18	4.39	20
network_patricia	1147	37	3.23	20
office_stringsearch	1086	27	2.49	120
security_blowfish	3655	32	0.88	1
security_rijndael	5904	38	0.64	1
security_sha	712	17	2.39	20
telecom_CRC32	196	7	3.57	40
telecom_adpcm	448	16	3.57	40
Parboil Benchmark Suit				
cutcp	3238	80	2.47	2
histo	1933	73	3.78	2
mri-gridding	2584	62	2.40	1
sad	2722	81	2.98	2
spmv	3931	82	2.08	3
stencil	1749	42	2.40	2
tpacf	1998	57	2.85	3

Table A1: LLVM IR Dataset used for training and evaluation. The first column lists the number of instructions traced and the second column shows the total number of conditional branch sites in each program. The third column lists the percentage of the instructions which is conditional branch. The last column gives the number of data sets we use to profile for each program.