Predicting Sequences of Optimization Passes using Machine Learning

Ashique Khudabukhsh (khudabukhsh@gmail.com)
Jayant Krishnamurthy (jayantk@cs.cmu.edu)

May 2, 2012

1 Project Description

The performance of code generated from an optimizing compiler depends not only on the optimizations performed, but also upon their order. However, it is typically unknown what sequence of optimizations should be applied to any given input program. This problem is exacerbated by the sheer number of possible sequences of optimizations, rendering infeasible a brute-force search for the best possible sequence. Selecting an appropriate sequence of optimizations is critical to achieving maximum performance in the compiled program.

Previously, machine learning has been used to predict a high-performance sequence of optimization passes for a given program or method[1, 2]. However, for computational tractability, these approaches severely limit the space of optimization sequences from which the prediction is drawn. In [1], the predicted optimization sequence must be a modification of an existing, hand-tuned strategy. In [2], a classifier is trained to predict an entire pass sequence; since a pass sequence is treated as an atomic unit, the predicted sequences must be relatively short (as the number of possible sequences grows exponentially in sequence length). As a consequence, these approaches are likely to find suboptimal optimization sequences, since they explore only a small subset of the space of optimization sequences.

We propose an alternative formulation of this pass-prediction problem: our approach independently models the behavior of individual transformation passes and the performance of compiled object files. The models learned in

this fashion can then be composed to predict the performance of a program when compiled with a sequence of optimizations. As a consequence, our approach can explore a much larger space of potential optimizations, while requiring a smaller number of machine learning models.

2 Method

Our optimization pass prediction approach has two phases. During the training phase, the approach uses sample programs, compiled with various optimizations, to learn the behavior of various compiler passes. This step learns how individual code transformation passes affect compiled bytecode. Additionally, we train a model to predict the running time of a program from its bytecode. This phase models how various aspects of compiled bytecode affect its running time. During the test phase, the approach predicts a good sequence of transformations for a given program. This phase performs a search over possible transformation sequences, using the predictions of the trained models to guide the search.

2.1 Training

The object of training is to produce two sets of functions, $f_p \in F$ and g. Each $f_p \in F$ represents the expected effect of transformation pass p; we represent programs as vectors in $\mathbf{R}^d$, hence $f_p : \mathbf{R}^d \rightarrow \mathbf{R}^d$. Training produces one function f_p for every possible transformation pass p; for example, we will learn one function for the $p =$`licm` pass, and an independent function for the $p =$`adce` pass. The function g maps program vectors to their expected running times, i.e., $g : \mathbf{R}^n \rightarrow \mathbf{R}$. This function is independent of any passes. Both functions are estimated from data, and, during test time, the combined effect of functions in f_p and g is used to evaluate the potential performance of a sequence of optimizations.

2.1.1 Generating Training Data

Both F and g are produced by training machine learning models on a training data set. We construct independent training data sets for each $f_p \in F$ and g. During this section, we assume access to a set of benchmark programs, from which all data sets are generated. See Section 3.1 for details on the benchmark data set.

The training data $D_p = \{(x^i, y^i)\}_{i=1}^n$ for pass p consists of tuples of compiled programs, (x^i, y^i), where y^i is generated by running pass p on x^i [1]. Given a set of benchmark programs, we create an x^i for each program by compiling it without optimizations using LLVM. We then generate y^i by compiling the same program using only optimization pass p.

Training data for the timing function g is generated similarly. This step aggregates all of the compiled programs generated in the previous step, then runs each program to estimate its running time. Two data points $(x^j, t^j) \in D_t$ are created for every $(x^i, y^i) \in D_p$, one by running x^i and one by running y^i. t^j is an estimate of x^j's running time in nanoseconds; we estimate t^j by running x^j 100 times and taking the mean.

In both cases, the compiled programs x^i and y^i are converted into feature vectors, as required by the machine learning algorithms.

2.1.2 Program Representation

Table 1 presents the features used to represent program bytecodes in the machine learning algorithms. For the instruction count features, we used the standard LLVM analysis pass `instcount` and the `STATISTIC` macro to gather data. We used a Perl script to parse the output into appropriate format.

In order to compute different features on variable counts, we modified the code of our homework assignment for Live Variable analysis and parsed the output using a Perl script into suitable format. For several loop-information-bases statistics, we modified our function-info code for homework assignment by combining LoopInfo analysis and iterating through the basic blocks present in each function.

2.1.3 Training Machine Learning Models

The data sets D_p and D_t are used to estimate the functions f_p and g. We experimented with two different machine learning algorithms for estimating these functions: k-nearest neighbors (KNN) and multivariate linear regression. However, we found that multivariate linear regression worked poorly on both tasks. Therefore, this section only describes the application of KNN to both the program transformation task and the timing prediction task.

[1] We use superscripts to index training examples, and subscripts to denote standard vector indexing.

Feature	Implementation
Number of basic blocks Number of Alloca instructions Number of Br instructions Number of FAdd instructions Number of FCmp instructions Number of FDiv instructions Number of FMul instructions Number of FPToSI instructions Number of FSub instructions Number of Load instructions Number of Ret instructions Number of Store instructions Number of memory instructions Number of Unreachable instructions	-instcount pass
Maximum number of variables in a function Maximum number of registers simultaneously live Total number of variable	Live variable analysis with further modification
Average depth of a statement inside a loop Maximum loop nesting depth Percentage of statements contained in some loop	FunctionInfo with LoopInfo

Table 1: Features used to represent the bytecode of compiled programs in the machine learning models. Each row of the above table is a dimension in the program's feature vector.

KNN is a simple machine learning algorithm that makes predictions for a test instance x based on similar instances in the training set $D = \{(x^i, y^i)\}$. Given the test instance, KNN identifies the k vectors $x^{i_1}, ..., x^{i_k}$ nearest to x according to some distance metric. It then returns the average of the labels for these points as its predictions, i.e., $y = \sum_{i=i_1,...,i_k} y^i$. An advantage of this procedure is that it does not assume a functional form for the prediction function, and thus is capable of representing extremely complex functions.

2.2 Predicting Pass Sequences

The trained models for predicting the effects of transformation passes ($f_p \in F$) and predicting program running times (g) are combined with a search procedure to identify good optimization pass sequences. The essential idea is simple – given a sequence of passes, say $\mathbf{p} = [p_1, p_2, ..., p_k]$ and an initial program x – we can predict the bytecode produced by compiling x with $\mathbf{p}$ by composing $f_{p_1}, f_{p_2}, ..., f_{p_k}$. We can additionally estimate the running time of the compiled program using g. Therefore, to identify a good pass sequence, we simply search over values of $\mathbf{p}$, constructing the final program and estimating its running time.

One problem with this simple strategy is that the number of passes $\mathbf{p}$ grows exponentially in the length of the pass. Therefore, we use a beam search to reduce search complexity. This search procedure maintains a list of the w best sequences found so far. At each step of the search, the algorithm attempts to extend each sequence by appending another optimization pass to the end. Each sequence produced in this fashion is evaluated by predicting the compiled bytecode (using the learned f_p functions) and predicting its running time (using g). The w sequences with the smallest predicted running times are retained for the next step of the search.

3 Experiments

3.1 Data Set

As our benchmark programs, we use the programs from the Computer Language Benchmarks Game [2]. These benchmarks are a set of relatively simple

[2] `http://shootout.alioth.debian.org/`

Distance Metric	1	2	3	4	5	6	7	8	9	10
Minkowski	7.46	7.65	6.47	5.86	5.55	5.51	6.30	10.66	18.82	27.27
Cityblock	4.64	5.72	7.07	7.03	6.99	7.22	7.57	9.62	17.63	23.86
Euclidean	4.91	6.01	7.41	7.97	8.31	8.75	9.38	10.93	18.80	24.92
Chebychev	7.72	10.28	9.17	7.99	8.33	7.89	8.08	15.52	25.05	32.14

Table 2: 10 fold cross-validation results for predicting program running time.

programs, constructed to compare the performance of algorithms across programming languages and platforms. An advantage of using this data set (over, say, SPEC) is that the programs are short; with larger programs, our simple representation of programs as vectors seems unlikely to accurately capture the program's intricacies. This data set can thus be seen as a sanity check for our approach. If our approach works on this data set, we can conceptually extend it to larger programs by independently predicting feature vectors for each function in the larger program.

3.2 Parameter Tuning

Broadly, KNN has two parameters – k, the number of neighbors, and the choice of distance metric. We used 10-fold cross-validation to find suitable values for these two parameters.

Table 2 shows the cross-validation results for runtime prediction. For a given entry, the row indicates the distance metric used and the column indicates the choice of k. Each value is the root-mean-squared error obtained when predicting running time. Based on this table, we selected k=1 and Cityblock as the distance metric for runtime prediction.

We did similar experiment to determine the parameters for predicting intermediate representations. The optimal parameter settings for each predictor are shown in Table 3. We found that k, when set to 2, consistently outperformed every other choice of k across every transform pass and any distance metric. However, for each individual pass, the choice of distance metric varied. We selected the "Minkowski" distance metric, as it was the best for a majority of the passes.

3.3 Predicting Pass Sequences

This experiment measures the end-to-end performance of our pass prediction system. We performed leave-one-out cross-validation on the benchmark

Pass	Best k	Best Distance Metric
adce	2	Minkowski
constprop	2	Minkowski
dse	2	Minkowski
indvars	2	Minkowski
inline	2	Euclidean
licm	2	Cityblock
mem2reg	2	Minkowski
sccp	2	Minkowski
tailcallelim	2	Minkowski

Table 3: Optimal KNN parameter settings for predicting intermediate representations, per pass.

training set. For each left out program, we used the remaining data to train both KNN models. We then used these models in the beam search procedure to predict a high-performance sequence of transformation passes for the left out program. This experiment simulates applying our technique to new programs.

We perform several versions of this experiment, varying both the length of the sequence of optimizations and the width w of the beam during beam search. Table 4 summarizes our results. Each row represents a single program compiled under 3 conditions, and each entry is the normalized running time of the program (where the unoptimized program takes unit time). The first condition is a baseline, LLVM with the `-std-compile-opts` flag. The next two pairs of columns represent two conditions for the beam search. Each pair of columns shows the normalized running time predicted by the machine learning models along with the true normalized running time for the predicted optimization pass sequence.

Generally, the results in Table 4 suggest that the default LLVM optimizations are quite competitive with the search procedure. However, there are a few programs on which the search procedure significantly outperforms the default optimizations (e.g., `binarytrees`). We can additionally determine that larger beam widths and longer optimization sequences improve the performance of the search procedure. Finally, comparing predicted runtimes with actual runtimes demonstrates that the machine learning models typically err on the optimistic side.

Program	LLVM Baseline	$w = 100$, $l = 10$ Predicted	$w = 100$, $l = 10$ Actual	$w = 10$, $l = 5$ Predicted	$w = 10$, $l = 5$ Actual
nbody	0.92	0.84	0.92	0.84	1.01
fibo	0.25	0.21	0.24	0.22	0.27
primes	0.89	0.96	1.03	4.39	1.08
algebraic	0.47	0.53	0.60	0.58	2.26
objinst	0.23	0.20	0.23	0.20	0.91
loop	1.10	0.98	1.14	1.06	1.72
echo	0.31	0.18	0.35	0.18	0.27
matrix	0.36	0.31	0.39	0.31	0.35
strcat	1.00	0.00	1.00	0.00	1.00
threadring	1.13	1.08	1.41	1.10	1.09
ary	0.72	0.58	0.67	0.58	0.63
heapsort	1.08	0.74	1.00	0.74	1.10
binarytrees	1.08	0.15	0.62	0.15	0.58
spectralnorm	0.98	0.00	0.99	0.00	1.00
methcall	0.28	0.25	0.28	0.25	0.26
lists	0.71	0.61	0.75	0.61	0.69
harmonic	0.86	0.79	0.97	0.79	1.07
recursive	0.74	0.03	0.52	0.03	0.53
except	0.20	0.22	0.22	0.22	0.89
fannkuch	0.57	0.32	0.33	0.32	0.43
takfp	0.28	0.24	0.33	0.24	0.68
strength	1.28	0.93	0.88	1.02	1.20
fannkuchredux	0.58	0.41	0.61	0.41	0.70
prodcons	0.68	0.49	0.76	0.58	0.72
nestedloop	1.13	0.70	0.79	3.23	0.96
meteor	0.82	0.00	0.94	0.00	0.89
partialsums	0.72	0.02	0.74	0.03	0.86
magicsquares	0.73	0.03	0.81	0.04	0.82
fasta	0.17	0.03	0.05	0.03	0.05
sieve	0.27	0.17	0.20	0.17	0.19
Average	0.68		0.66		0.81

Table 4: Prediction results for the beam search task. Each entry is a normalized running time, where the unoptimized version of the program is assumed to run in unit time.

4 Discussion

Predicting the best performing sequence of optimizations is critical in order to achieve maximum program performance. We presented a novel machine learning approach to this task that can predict arbitrary long optimization sequences. Our results suggest that this procedure works much better than the default optimizations for some programs, but on average has similar performance to the default settings.

5 Distribution of Credit

We equally distributed the work for this project. Jayant did the program timing and beam search, and Ashique generated features and trained the machine learning models.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. G. Stoodley. Using machines to learn method-specific compilation strategies. In *CGO*, pages 257–266. IEEE, 2011.