Building Fast Performance Models for Loop-Free 64-bit x86 Code Sequences

Final Project Report
CS229: Machine Learning, Autumn 2013
Stanford University

Negar Rahmati
Stanford University
rahmati.nr@gmail.com

Jessica Su
Stanford University
jtysu@stanford.edu

Judson Wilson
Stanford University
judsonw@stanford.edu

1. INTRODUCTION

Ever since the advent of the compiler, computer scientists have striven to improve the efficiency of their resulting programs. Superoptimization is an active research area in this field, where researchers seek methods of finding optimal instruction sequences to achieve arbitrary tasks. Researchers Schkufza and Aiken have proposed a method called STOKE, or Stochastic Superoptimization, that uses a Markov Chain Monte Carlo method as the basis of the optimization algorithm [2]. The method requires predicting runtimes of many alternative instruction sequences to search for possible improvements.

Determining the relative efficiency of various code sequences is a nontrivial task. Modern x86 processors have very complex architectures with hundreds of instructions. To boost performance, processors incorporate many hardware optimizations such as pipelining, adding ALUs, register renaming and out-of-order execution [3]. The machinery is quite complicated, and while the operation of the machine is deterministic, to an observer the operation exhibits unpredictable (but repeatable) performance depending on the sequencing of the instructions and the beginning state. To accurately predict execution time would require analysis of proprietary engineering information, or prohibitively complex brute-force testing and modeling.

In this paper we use machine learning techniques to create assembly instruction runtime prediction models. We experiment with different features and algorithms, including linear regression, support vector regression, and gradient descent. The goal is to produce a more accurate model for STOKE that maintains high prediction speed. The purpose of the model is to gauge relative performance between code sequences, such that a set of possible alternative code sequences can be generated. Estimation methods that produce results with higher accuracy at the expensive of speed can be used to refine the resulting set, a topic beyond the scope of this paper.

2. METHODS

2.1 Data Collection

The main data set we used was provided by Eric Schkufza, our collaborator at Stanford University. He generated instruction sequences for an Intel Nahalem family processor, measured runtimes on a processor in this family. The data consists of randomly generated instruction sequences of length 1 to 16. It contains 100000 sequences of each length, together with their runtimes. We chose at random 70% of the dataset to use as training data, and 30% for testing.

As a starting point, we chose a large subset of the available instructions, with the notable exception of operations that branch. Our dataset contains 506 different instructions. We also chose to limit the operands to registers and literal values only, to prevent behavior related to external memory access. The dataset was designed to narrow the problem down to an investigation of the interaction of instructions proceeding linearly through the program. It is free of external factors with large influence on the runtime.

The runtimes were calculated by placing the sequences in a program that looped over the sequence 10^5 times and compared the timestamp at the start and end. The programs were run in a Linux environment, and the timestamps were calculated relative to the execution time of the process, excluding time spent in other threads. The measurements include the runtime of the looping instructions, as well as a series of `nop` instructions that were used to help reduce any interaction between the loop instructions and the instructions under test. We model the effect of these loop and `nop` instructions as a constant offset across all test results. We measured this value by testing zero-length instruction sequences, and subtracted it from the runtimes.

In order to extract the dependencies between instructions, a topic discussed later, we used the x64asm library. x64asm is a c++11 library by Eric Schkufza for working with x86_64 assembly that provides a parser, an in memory assembler, and primitives for building data flow analysis. The library includes tools for extracting the read and write register dependencies for instructions and their operands. We wrote a program to generate a derived feature set from the set of instruction sequences.

2.2 Models and Learning Algorithms

All the models we used are linear predictions, where various features contribute to the predicted runtime linearly. The measured runtime of a code sequence, or program, is represented by y, and our prediction is represented by $\widehat{y}$. The prediction is the linear combination of features x and their associated runtime contributions θ:

$$\widehat{y} = x^T \theta$$

For all models we used least squares regression to find the values of parameter θ that minimized the sum of squares of error for the

training data set:

$$\theta_{opt} = \arg\min_{\theta} ||y - X\theta||_2^2$$

Specific methods of solving the regression problem are discussed with each model.

In addition to least squares regression, we also attempted to use support vector regression, using the libsvm software package [1]. The hope was that it would provide the ability to use non-linear functions of our features. However, we found the software to be much too slow to be useful for our dataset.

2.3 Model Notation

The aim of this project is to model the runtime of arbitrary sequences of machine instructions, or programs. Let program p be an ordered set of program lines $(p_1, p_2, p_3, \ldots, p_k)$, where each line of the program p specifies an instruction from the instruction set $\mathcal{I} = \{\psi_1, \psi_2, \ldots \psi_\ell\}$, and associated operands. Denote the length of a program p by $|p|$.

Let $x^{(i)}$ be the feature vector derived from the ith program $p^{(i)}$. Our target variable $y^{(i)}$ is the measured runtime of 10^5 iterations of $p^{(i)}$. Because the program runtime may change with CPU register data and other conditions, and because the test measurement system is not perfect, $p^{(i)} = p^{(j)}$ does not necessarily imply $y^{(i)} = y^{(j)}$.

2.4 Static Feature Model

We started with the natural approach, replicating the model used by Schkufza and Sharma. This static feature model assumes the runtime of a line p_j is a static property of the corresponding instruction ψ, independent of the operands, the processor state, or other program lines $p_k : k \neq j$. Therefore, the runtime of a program p will be the sum of the runtimes of the instructions ψ assigned to each line p_j. We define feature $x_j^{(i)}$ as the number of times instruction ψ_j appears in program $p^{(i)}$:

$$x_j^{(i)} = \sum_{k=1}^{|p^{(i)}|} 1\{p_k^{(i)} = \psi_j\}$$

This model would accurately describe a simple computer that lacked hardware optimization, where resource access times are constant, and the instruction set is very simple.

The original example runtimes $y^{(i)}$ include a very large amount of overhead for looping that are not a property of the program under test, $p^{(i)}$. We model this time as a constant offset to the runtime, because it is a consistent sequence of instructions meant specifically to not interact with the program p. We measured this time directly with a zero-length program, and found the results to be consistent across many tests. We found that calculating the value using an intercept term in our model did not meaningfully change the test error. Therefore we subtract the empirical offset from the runtimes $y^{(i)}$ for this and all subsequent models to reduce computation complexity.

The runtimes θ_j for each instruction were learned using least squares regression. We solved the least squares error minimization problem $\arg\min_\theta ||y - X\theta||_2^2$ using the MATLAB matrix left division operator, which is only possible for our dataset by taking advantage of the sparsity of X. In our dataset, only 16 of the 506 features can be nonzero.

2.5 Pairwise-Neighbors Feature Model

A logical method of improving the Static Model is to add some dependence between neighboring instructions. For example, perhaps if instruction ψ_i follows ψ_j the processor can use some optimization to reduce the execution time, such as performing two multiplications in parallel. There are many ways to accomplish this goal, but perhaps the simplest is to count the occurrences of ordered pairs of instructions in the sequence:

$$\hat{x}_{m,n}^{(i)} = \sum_{k=1}^{|p^{(i)}|-1} 1\{p_k^{(i)} = \psi_m \wedge p_{k+1}^{(i)} = \psi_n\}$$

Let the feature vector x contain all the static terms from before, and all $|\mathcal{I}|^2$ features $\hat{x}_{m,n}^{(i)}$.

This model proved to be too memory intensive for MATLAB's matrix left divide operator. Although sparse, the model introduced $\sim 2.5 \times 10^5$ new features. An iterative approach was needed. Note the normal equations can be formed into a standard solution of linear equations problem:

$$\begin{aligned} (X^T X)\theta &= X^T y \\ A\tilde{x} = \tilde{y} \quad &: \quad A = (X^T X) \\ &\quad\ \tilde{x} = \theta \\ &\quad\ \tilde{y} = X^T y \end{aligned}$$

This can be solved using various algorithms. We chose the Biconjugate Gradient Stabilized Method, Bi-CGSTAB [4]. Special care must be taken to ensure sparsity is preserved. $X^T X$ is not necessarily a sparse matrix (it is likely a ~ 250000 by ~ 250000 dense matrix), so $X^T X z$ should be computed by multiplying the sparse matrix X^T with the vector Xz.

In general, this method proved to be more practical than matrix left division, so it was used for all further models. 1000 iterations of Bi-CSTAB were used unless the software detected that further progress was inhibited by numerical precision.

2.6 Two Stage Regression by Variance Class

The Pairwise-Neighbors feature model introduced $\sim 2.5 \times 10^5$ new features to the model, and as a result suffered from overfitting (as will be discussed in the Results and Discussion section). Upon inspecting the behavior of various examples from our dataset, it was determined that the presence of some instructions lead to larger observation variance than others, and thus increase the susceptibility to overfitting.

By inspecting various programs of length 1, it was observed that some instructions tend to have a predictable runtime with variance much smaller than the mean, similar to a narrow Gaussian or Poisson distribution. Other instructions tend to exhibit strange runtime distributions. Many appeared bimodal or exponential in nature. Literature suggests that some of the bimodal-time instructions may be explained by error conditions that trigger on bad input data. Other distributions may be due to instructions that perform calculations with iterative algorithms, such as instructions that calculate a square root. As such, we will classify the instructions as "low-variance" and "high-variance" for this model.

Ideally we would normalize the data in such a way to produce a BLUE estimator, but that would require detailed variance measurements. Modeling the variance of every instruction is in general

a difficult problem, and modeling the variance of each instruction when combined with other instructions may not be possible.

In the interest of simplicity, the parameters and examples were partitioned by their variance class and regression was used to solve for the parameters of the two classes in two stages. Consider replacing our original error minimization problem with the following equivalent formulation, permuting the columns of X and rows of θ according to some classification lv (low variance) and hv (high variance), and then splitting their product:

$$X = [X_{\text{lv}}|X_{\text{hv}}], \quad \theta = (\theta_{\text{lv}}, \theta_{\text{hv}})$$
$$\arg\min_{\theta_{\text{lv}},\theta_{\text{hv}}} ||y - X_{\text{lv}}\theta_{\text{lv}} + X_{\text{hv}}\theta_{\text{hv}}||_2^2$$

Then permute the rows of X and y identically to produce the following form:

$$y = \left[\begin{array}{c} y_1 \\ \hline y_2 \end{array}\right], \quad X = \left[\begin{array}{c|c} X_{\text{lv1}} & 0 \\ \hline X_{\text{lv2}} & X_{\text{hv2}} \end{array}\right]$$

This formulation suggests a method for decoupling the low-variance and high-variance instructions, such that the high variance terms do not dominate the low variance terms:

$$\theta^*_{\text{lv}} = \arg\min_{\theta} ||y_1 - X_{\text{lv1}}\theta||$$
$$\theta^*_{\text{hv}} = \arg\min_{\theta} ||y_2 - X_{\text{lv2}}\theta^*_{\text{lv}} - X_{\text{hv2}}\theta||$$

This is a two stage optimization problem, first solving θ^*_{lv}, then θ^*_{hv}. The first stage estimates only the low-variance feature parameters, by exploiting training examples that do not involve high-variance features. The second state solves for the high variance feature parameters by minimizing the error of the remaining examples, whose variance is likely dominated by these terms.

Finally, to implement the model requires a method of classifying features as low-variance or high-variance. To do this, a relative measure of variance was developed for the underlying instructions. For the set of single line programs $P_1 = \{p^{(i)} : |p^{(i)}| = 1|\}$, sets of runtime samples for each instruction ψ_j were collected:

$$Y_j = \{y^{(i)} : p_1^{(i)} = \psi_j, p^{(i)} \in P_1\}$$

Then the relative variance of an instruction ψ_j was defined as the ratio of the standard deviation of Y_j to the mean of Y_j. A histogram of the relative variances of all instructions appearing in P_1 shows a bimodal distribution with a large gap. See Figure 1. A threshold of 0.7 was chosen from this gap to serve as a classifier, where an instruction is considered high or low variance if its relative variance is above or below this threshold, respectively.

Extending the definition from instructions to the static features is straightforward: a feature x_j is low-variance if ψ_j is low variance, and high-variance if ψ_j is high-variance. The classification of the pairwise-neighbor features is slightly more complex. $\hat{x}_{i,j}$ is high-variance if ψ_i or ψ_j is classified as a high-variance instruction, otherwise it is low-variance. (Relaxing this definition to dependence on strictly ψ_i or strictly ψ_j may be an interesting area for further investigation). Note that this definition rules out a large number of rows for inclusion in X_{lv}, because the probability of an n-instruction program not having any high-variance instructions decays with ϕ^n, where ϕ is the Bernoulli probability of an instruction being low-variance. As a result, the testing and training set X_{lv} tend to have a sampling bias towards shorter programs, and X_{hv} towards longer programs. Custom datasets could be generated to account for this.

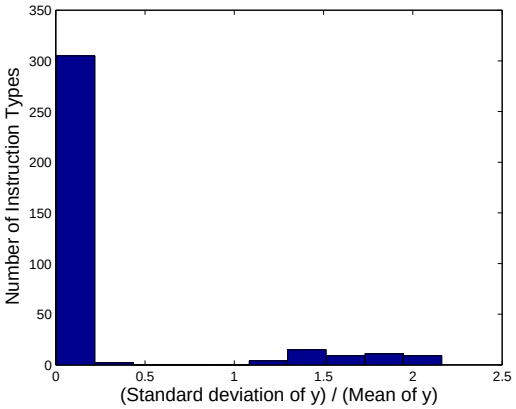

Figure 1: Relative Variance Histogram

2.7 Dependency Graph Based Features

The designs of modern x86_64 processors contain elaborate hardware to locally parallelize instruction sequences. Using techniques such as register renaming (where general registers are used to create an instance of a register used in a program), multiple independent subsequences of code may be computed at the same time. This technology is called out-of-order execution. [3] Clearly such behavior is non-linear with respect to the feature sets defined thus far.

Listing 1 A Toy Program

```
1:  movl %eax, %ebx
2:  rclw $0x1, %ax
3:  movl %ebx, %ecx
4:  movl %ecx, %edx
5:  movl $0x10, %ecx
```

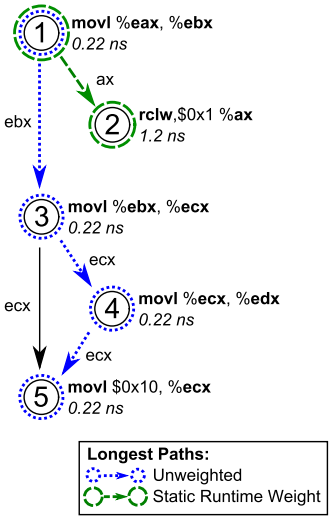

Figure 2: Dependency Graph

We experimented with a few new feature sets with the hope of capturing some of this behavior in a linear model. To do this, we created a new model of the processor such that it parallelizes and schedules all instructions to execute as quickly as the required register state is valid (including those registers named in the operands, and those not named in the operands, such as flag registers). For example, in the toy program listed in Listing 1, instruction 3 must

Table 1: Training and Testing Errors For Models Tested.

Model / Features	Number of Feat.	**Training Error** All Programs		**Test Error** All Programs		**Test Error** Single Line Programs		**Test Error** Low Variance Only	
		RMS	Rel	RMS	Rel	RMS	Rel	RMS	Rel
Baseline ($Y = 0$)	0	2.24×10^6	1	2.25×10^6	1	7.06×10^5	1	3.72×10^5	1
Static Instruction Features	506	1.67×10^6	2.50	1.68×10^6	2.49	6.07×10^5	3.35	1.30×10^5	0.99
Static + Pairwise Inst. Features	256542	1.20×10^6	0.007	1.71×10^6	4.24	6.10×10^5	3.97	5.74×10^5	5.01
2-Stage Static + Pairwise	256542	1.31×10^6	2.49	1.69×10^6	2.99	6.07×10^5	2.50	1.25×10^5	0.68
Static + Unweighted Height	507	1.66×10^6	2.49	1.67×10^6	2.46	6.08×10^5	3.31	1.28×10^5	0.94
Static + Weighted Height	507	1.66×10^6	2.50	1.67×10^6	2.47	6.08×10^5	3.30	1.30×10^5	0.98
Static + Longest Path Static Inst.	1012	1.66×10^6	2.67	1.67×10^6	2.64	6.16×10^5	4.70	1.60×10^5	1.63

be executed after instruction 1, since both instructions use register `%ebx`, so instruction 3 depends on instruction 1. However, instructions 3 and 2 do not affect any of the same registers, so they can be executed simultaneously.

To represent these dependencies, we used an idea proposed by Eric Schkufza. We draw a dependency graph (Figure 2), where (i, j) is an edge in the graph if instruction j depends on instruction i. We conjectured that the following features might affect the runtime of the program, starting with the simplest:

2.7.1 Unweighted Dependency Graph Height

We define a feature equal to the height of the unweighted dependency graph (plus 1), i.e. the number of instructions on the longest path. In Figure 2, this feature would be 4, corresponding to the longest path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$. This number is a single additional feature used along with the static features described previously. This is perhaps an overly simple model, but we believe the feature is informative and thus may lead to better predictions.

2.7.2 Weighted Dependency Graph Height

We define a feature equal to the sum of the runtimes of the instructions on the longest-running path, i.e., the path with the greatest total runtime. In Figure 2, this feature would be 1.42 ns, and the longest path would be $1 \rightarrow 2$. We describe this as a weighted path, where each instruction node is weighted by the static runtime of the instruction. We estimated these instruction runtimes using our results from the Static Feature Model. This method assumes that execution times of the instructions themselves do not change based on ordering of the instructions. We view this as an enhancement to the simple unweighted height feature just described. For a processor that acts as our model predicts, we would expect that this feature would contribute greatly to the runtime, and would predict perfectly if the runtime for each instruction were known perfectly when creating the feature.

2.7.3 Longest Dep. Graph Path Static Inst. Features

We define one feature per instruction in $\mathcal{I}$, equal to the number of times each instruction appears on the longest-running path. In Figure 2, the `movl` feature would have value 1, the `rclw` feature would have value 1, and all other such features would have value 0. This model extends the previous model by not assuming we know the runtime for each instruction, except for the purpose of identifying the longest path during feature calculation. (i.e. it is possible that a wrong estimate will choose feature values corresponding to the wrong path.) For a processor that follows the model, assuming we have properly determined the longest path, this model should learn the runtimes for each instruction. The static features described earlier were also included, for consistency. In the ideal case the static feature parameter would be zero, and these new longest-path static feature parameters would be the average runtimes for each instruction. This model is the natural extension of the static feature model to the case of perfect out-of-order execution.

3. RESULTS AND DISCUSSION

For each model, we measure the prediction error on the training and test sets. For further insight, we also calculated the prediction error on two subsets of the testing data: the examples that contain programs of length 1, and the examples whose programs contain only the low-variance instructions (as described in Section 2.6). We define the error in two ways. The first is the RMS error:

$$E_{\text{RMS}} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \widehat{y}^{(i)})}$$

where m is the number of data points. The least squares objective of linear regression, as used for all of our models, equivalently minimizes this error. We also used a form of the average relative error, for ease of reading:

$$E_{\text{Rel}} = \frac{1}{m}\sum_{i=1}^{m}\left|\frac{y^{(i)} - \widehat{y}^{(i)}}{y^{(i)}}\right|$$

The results for the various models described above are listed in Table 1. We have also included an absolute error of a prediction of $\widehat{y} = 0$ to serve as a baseline to compare to the RMS error of each model.

3.1 Static Feature Model Analysis

In terms of RMS error, the Static Instruction Features proved to be a better predictor of runtime than the baseline, as one would expect. Least squares regression directly reduces this cost. Thus the model has made a small improvement. Further, the Training and Testing error for the "All Programs" subsets are nearly equal, suggesting that overfitting is not a problem. The relative errors are all either multiples larger than 1, or nearly 1, suggesting that the least squares objective may be weighting outliers too high for some purposes.

3.2 Pairwise-Neighbors Model Analysis

The addition of the Pairwise-Neighbors Features reduced the training error. However, the RMS and relative test error increased for all test sets. Clearly the addition of this large number of features is causing overfitting. Somewhat amazingly, the relative training error was reduced by two orders of magnitude. This suggests that there are relatively small number of statistical outliers that are severely

affecting the RMS error, due to the squaring effect. That said, overfitting is a real issue, so this result is only mildly useful. These observations led to the development of the Two-Stage Regression algorithm, introduced in Section 2.6.

3.3 Two Stage Regression Analysis

Two-Stage Regression produced slightly increased training error over standard regression results for the Static and Pairwise Model, but this was also expected because the standard least-squares regression solution is the minimizer, by definition. So if there is any improvement, it would be in the test error.

The method produced similar results to the Static Instruction Features Model for all three types of RMS test error. The relative error was significantly different, notably lower for the Low Variance and Single Line Program subsets. This is likely due to the first optimization step which minimizes prediction mean square error on the low-variance training set.

3.4 Dependency Graph Related Results

All three models using the dependency graph related features produced results nearly identical to the Static Instruction Features Model, with slightly increased test error for the Longest Dependency Graph Path Static Instruction Features Model. This model doubles the number of features over the Static Instruction Features, and thus the result appears to suffer from overfitting.

4. CONCLUSIONS AND FUTURE WORK

Generalizing the results, none of the models made a worthwhile improvement over the Static Instruction Features Model. The Two-Stage Regression method applied to the Static and Pairwise-Neighbors Feature Model may have yielded some improvement, but at the cost of adding several orders of magnitude more features. It remains to be seen if the result is real, and if the added complexity will make the improvement useful for the intended purpose - producing a model that can rapidly estimate runtime.

It is likely that applying the two-stage regression technique to the other feature sets may yield similar improvements in terms of some of the test sets. The initial optimization step tends to more tightly fit the low-variance instructions, and should reduce testing error on subsets with fewer high-variance featured programs. Performance on the full test set will likely change little, as the high variance instructions (which likely dominate error) should be little affected.

This theory suggests that, in general, more information about the variance of the data is needed to make improvements. The variance likely has links to the individual instructions, as well as to their interactions, and the initial data in the registers when the execution time test is run during dataset generation. With adequate knowledge a BLUE estimator could be deduced. On the other hand, perhaps the high variance instructions could simply be removed, to reduce skew caused by outliers. In fact, the two-stage regression techniques could be extended to multiple stages, and a new dataset could be generated to provide the desired number of examples for each stage.

It is somewhat interesting that the dependency tree based models did not yield improvement. The authors believe the models should yield better results. It is unclear whether the models are flawed, or if there are problems in the test data, the feature generation, or the regression solver.

If using the same example data, the next steps are to explore the possibilities of removing examples with instructions that are problematic, similar to what was done in two-stage regression, but focusing the analysis on these well-behaved instructions.

If more data can be generated, one could attempt to derive more information related to the mean and variance for each instruction in isolation, and more closely examine the behavior of relatively short programs with only low-variance instructions. This may yield clues about the performance of the models.

To reduce the noisy nature of the dataset, a different approach could be used to generate examples. High level tools exist to give heuristic prediction data for program runtimes, such as the Intel Architecture Code Analyzer tool. These tools are too slow to use in the stochastic optimization setting, but could be used for model generation. Instead of running the instructions many times in a loop to measure runtime, one could generated example runtimes using one of these tools. The resulting examples could potentially yield better results when learning the parameters of the models discussed in this paper, assuming the tools yield data that is more consistent for constant input.

5. ACKNOWLEDGMENTS

We would like to thank Eric Schkufza for the project idea, for generating the dataset and providing useful tools and information, and for providing the basic underlying idea behind our dependency graph models.

6. REFERENCES

[1] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machine.

[2] R. S. E. Schkufza and A. Aiken. Stochastic superoptimization.

[3] D. C. Eric Sprangle. Increasing processor performance by implementing deeper pipelines. *Computer Architecture*, pages 25–34.

[4] H. A. van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems.