Building Fast Performance Models for x86 Code Sequences

Justin Womersley JWOMERS@STANFORD.EDU
Stanford University, 450 Serra Mall, Stanford, CA 94305 USA

Christopher John Cullen CJC888@STANFORD.EDU
Stanford University, 450 Serra Mall, Stanford, CA 94305 USA

1. Introduction

STOKE is an aggressive low level compiler which generates complex assembly level optimizations using random search. Part of this process includes estimating the performance properties of short 64-bit x86 instruction sequences. In this project we aim to train a machine learning model to more accurately estimate the performance of these instruction sequences. The predictor has to be as accurate as possible (given that CISC architecture is extremely complex) while also being able to sustain a throughput of 1M predictions/second to remain useful within STOKE optimization process.

2. Background, Scope & Requirements

Current performance estimations for STOKE are simply a sum of average run times for each instruction. This is fast, but not very accurate. On the other side of the spectrum are cycle accurate simulators, which give accurate preformance estimations but at the cost of very long run times. Thus our goal is to achieve an estimation model that is both more accurate than the current sum-of-averages predictions while also fast enough to be useable in the STOKE compiler.

To maintain feasibility, the scope of this project has been limited to short code sequences of 20 instructions or less. In addition, certain rare and more complex x86 instructions, like those including memory access, are not required in the model, and thus not included in the training or testing data sets.

3. Data & Features

Our data is in the form of short x86-64 loop-free code sequences along with their known execution times. For each sequence length of 1 to 16 instructions, we have 100,000 randomly generated sequences, each of which has been executed 1,000,000 times in a loop to provide an accurate estimate of average execution time. This includes padding each sequence with 20 ”nop” instructions in an attempt to minimize more complex pipeline and caching effects. This data is then stored as an xml database with code sequence and execution time. Using the starter code from the x64asm tools, we created a parser to read the xml entries and generate feature vectors for the entire data set. This process is shown in figure 1.

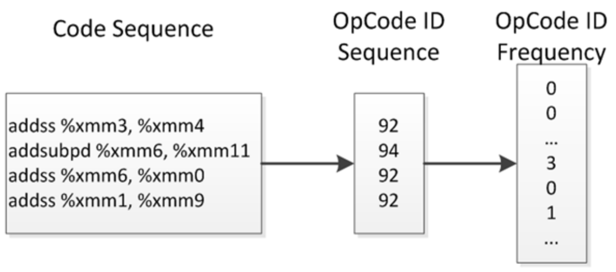

Figure 1. The process of extracting the opcode data into a sparse opcode ID vector. Each sequence gets translated into 3800 length vector, one entry for each opcode.

We then took a sample of 10% of the generated feature vectors (160,000 examples), and split that set into 90% training (144,000 examples) and 10% test (16,000 examples). Both training and testing sets were then radomized to more realistically represent real world data. Each of our chosen models were then trained and evaluated in MatLab.

Top performing models were evaluated more thoroughly by partitioning the test set by sequence length and testing the models on each partition. The best performing model was implemented in C++ to measure the prediction runtime in order to validate our throughput requirement. For a complete overview of our data preperation, modeling and testing process see figure 2.

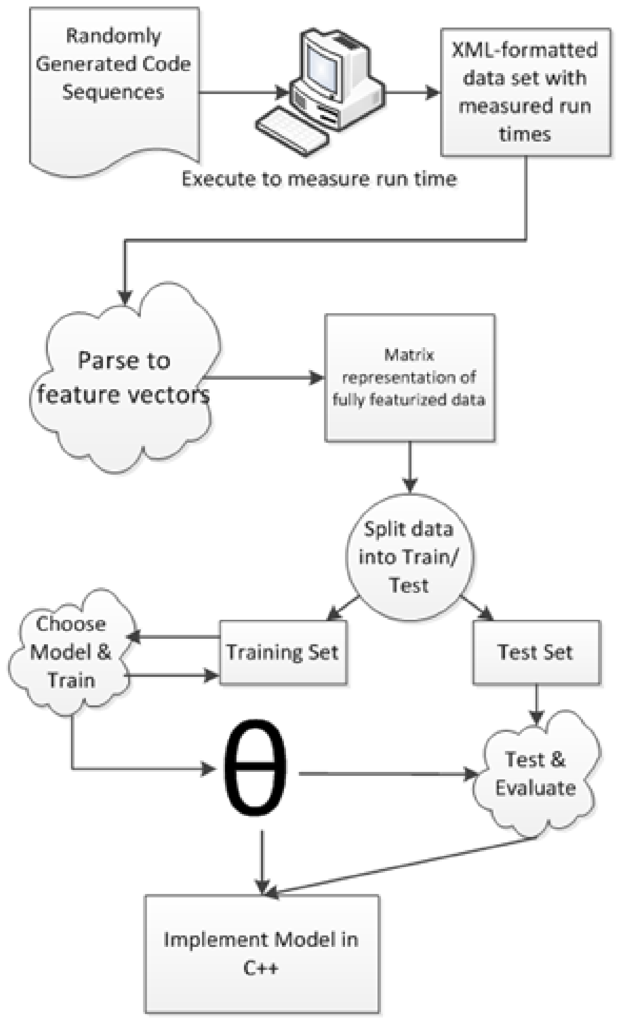
Randomly Generated Code Sequences
Execute to measure run time
XML-formatted data set with measured run times
Parse to feature vectors
Matrix representation of fully featurized data
Split data into Train/ Test
Choose Model & Train
Training Set
Test Set
θ
Test & Evaluate
Implement Model in C++

Figure 2. Process for generating, preparing, modeling and testing on our data

4. Data Issues

4.1. Data Variability

Prior to defining features, we performed data analysis on our generated data, examining means and variances over the different sets. Execution times were mostly linear with the number of instructions in the sequence, but extreme outliers raised the mean of the dataset to be non-representative of the true tendency of the data. As seen in figure 3, the vast majority of execution times were in the sub-1,000,000 range, sloping gradually above 1,000,000 for code sequences of length > 10. An optimal predictor would need to be able to accurately model both the lower execution times of the bulk of the data and the higher (by an order of magnitude) execution times of the minority outliers. Figure 4 shows (with logarithmic y-axis) the distribution of execution times in sequences containing only one instruction. 93.7% of sequences are $< 250,000$ and 97.7% of sequences are $< 500,000$, but one sequence had a runtime of $> 12,000,000$. Thus the variability of even the simplest of our data creates a difficult modeling problem and any successful model would have to accurately predict these outliers.

Data #	Opcode	Registers	Runtime
1-89832	SQRTPD	%XMM15, %XMM13	219,647
1-17062	SQRTPD	%XMM15, %XMM13	1,007,578
1-20519	SQRTPD	%XMM15, %XMM13	6,994,932

Table 1. Three different run times for the same opcode and register sets indicate the huge variance in our data due to unmodeled effects. Note these runtimes were achieved with only the 1 instruction pre-padded with 20 ”nop” instructions.

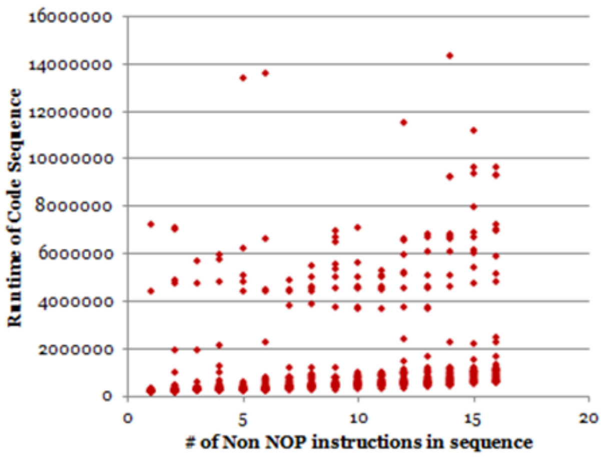

Figure 3. The distribution of runtimes for varying sequence lengths. The vast majority of runtimes were ¡ 1,000,000 with a clear linear trend for these non-outliers. See figure 4 for a more accurate percentage breakdown or runtimes for sequences of 1 instruction.

4.2. Unmodeled effects

Along with high variability seen within a minority of code sequences of the same length, there were several cases where high variability was seen within different executions of the same sequence. Table 1 shows three separate examples of the same code sequence, with drastically different execution times. Whether due to latent processor state or other factors that our data do not capture, this discrepancy appears only as noise in our data. Because of this, no predictor can (deter-

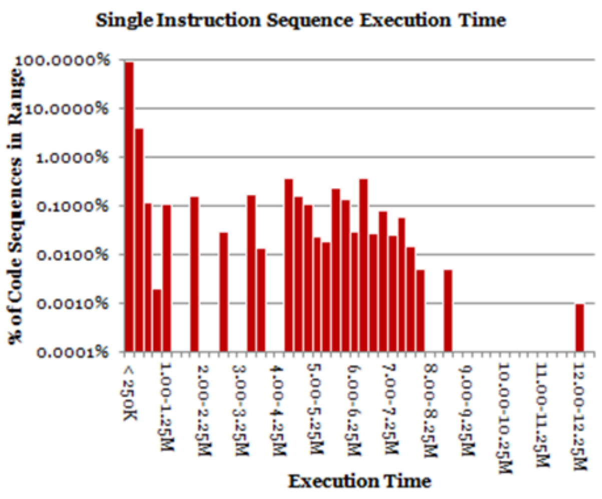

Figure 4. Execution time for single instruction sequences - note that almost 98% of the runtimes fall below 500,000. The remaining 2% of outliers take on values of as much as 12,000,000.

ministically) perform perfectly on our data set. An optimal predictor can only minimize error over ostensibly equivalent data.

5. Prediction Models

Feature Creation

Calculating runtime is at its most basic level a sum-of-parts problem with the opcodes as the main contributing factor. Thus our first step was to map our sequences to an opcode frequency feature matrix. Since we had between 1 and 16 opcodes per sequence with 4000 opcodes, this produced a very sparse feature matrix.

Linear Regression

Due to high throughput requirement, we could not use any form of non-parametric algorithm as prediction throughput with such algorithms would not be high enough. Fortunately though, this strict speed requirement did not extend to training our model, allowing us a larger breadth of training approaches and implementations in MatLab.

A linear regression with $h(x) = \sum_1^n \theta_i x_i = \theta^T x$ using the normal equations quickly ran into two issues: running out of memory for even moderate data sets, and getting non-invertible matrices dues to the extreme sparsity of our data set. Since our design matrix X is a very sparse matrix, using the normal equations was simply not feasible.

Modified Linear Regression

Using a stochastic gradient descent on our training set achieved good results with a small training error. However the feature weights of θ often ended up negative for opcodes that did not appear often in the training set. While this is mathematically valid - there is no logical mapping for this to instruction runtime. Since we know all instruction runtimes must be non-negative, we amended the stochastic gradient descent update step to reflect this prior knowledge to the following:

$$\theta_j := max(0, \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)})$$

which we will refer to as the θ-floor update. In addition, the outliers described in figure 3 often resulted in large oscillations as theta was updated to reflect these outliers. As a result we also reduced our learning rate α to better cope with this. This change, in addition to including our prior knowledge that θ_j's cannot be negative, reduced the oscillating effect we encountered when predictions are far from $y^{(i)}$ and resulted in no negative feature weights for θ.

Outlier Exclusion

Since the the small minority of extreme outliers, including known unmodeled effects described in table 1, may impede the the regression to model the more linear sequences, another approach we took was to train the regression model with a dataset that excluded any extreme outliers. The aim was to better model the vast majority of the data at the expense of badly predicting the $< 10\%$ of outliers and those with unmodeled effects. For each of the 1-16 training sequences of 100,000 we removed 30% of training examples that were lying furthest from the median and then implemented the stochastic gradient descent using our θ-floor update modification.

Additional Features

Lastly, to better predict the outliers we separated them out and analyzed which, if any, opcodes and register combinations resulted in these large runtimes. Once we had a subset of 44 opcodes and 16 indicative registers, we added these additional 256 features (cross product for register pairs of each of the 16 registers for any of the 44 opcodes) to our sparse opcode feature matrix in the hope of modelling these more complex runtime results.

6. Evaluation Metrics

The baseline benchmark against which we evaluated our predictions was the runtime predictor that is currently being used in STOKE. This is a simple table lookup using Agner Fog instruction tables [2] for the Haswell architecture. To evaluate our models and predictions against this baseline benchmark we used two different error metrics:

$$\text{average error} = \varepsilon_a = \frac{\sum_{i=1}^{m} \mid y^{(i)} - h(x^{(i)}) \mid}{m}$$

where m is the number of testing examples, y is the actual runtime and h(x) is the predicted runtime using our feature vector x.

$$\text{threshold error} = \varepsilon_t = \frac{\sum_{i=1}^{m} 1\{\mid y^{(i)} - h(x^{(i)}) \mid > \delta_t\}}{m}$$

where δ_t is an acceptable error margin, in our case 30% or 0.3. Since our extreme outliers could easily skew the average error even in a reasonable good predictor, the threshold error would in this case represent a more useful error metric as it reports the percentage of predictions that were wrong.

To evaluate our timing requirement we developed our final predictor in C++ and timed it making 100,000 predictions, ensuring that it met our throughput requirement.

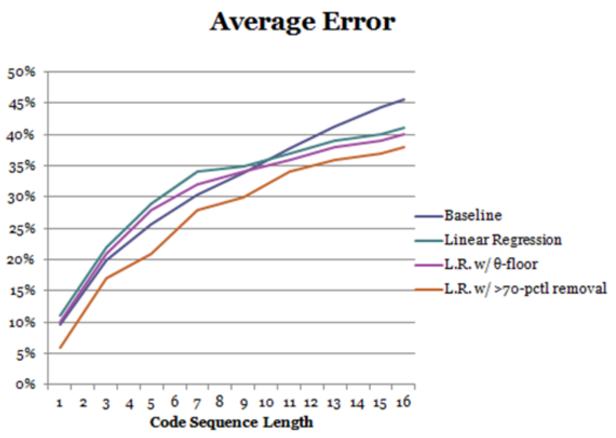

Figure 5. Average error for different models. The best performing model (lowest error) was the linear regression with the outliers removed during training.

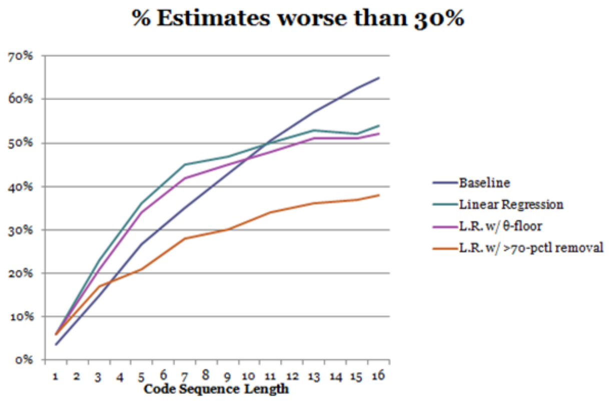

Figure 6. Threshold error for different models. The best performing model (lowest error) was the linear regression with the outliers removed during training.

7. Results

7.1. Accuracy & Error Rates

Our best performing model was the modified linear regression using θ-floor trained on our dataset that excluded the outliers - it was able to improve the average error by 16% from the baseline benchmark, and improved on the baseline's threshold error by 29%, while also maintaining the necessary throughput. Figure 7 shows the overall errors for our three top-performing models, along with the baseline. Figure 5 and figure 6 show the results for each of our error metrics, partitioned by sequence length.

Linear regression and θ-floor linear regression on Opcode frequency were both outperformed by the baseline overall, but had better error rates than the baseline as sequence length increased. This suggests that they are useful for predictive qualities on larger code sequences. Linear regression with advanced features such as register interaction modeling are not included in these results because they performed worse than the baseline in both metrics for all sequence lengths. Any attemps we made to model the outliers using these register features only worsened the predictive power of our model. Linear regression on Opcode frequency with outlier removal performed best overall, and outperformed the other models in almost every metric.

7.2. Timing & Throughput

We implemented our final model as an iterative table-lookup sum computation in C++, achieving an average throughput of 1.37M predictions per second on code sequences of length 16. This throughtput meets our throughout requirement. In addition, converting our final theta from floating point values to fixed point values would increase the throughput even further while having very little effect on prediction accu-

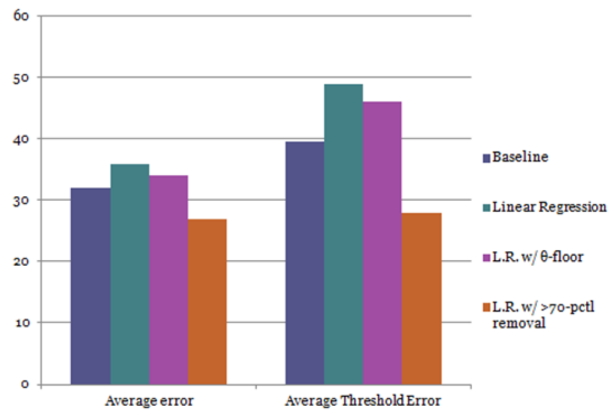

Figure 7. Overall errors for our different models. Both the standard linear regression and modified θ-floor regression were not able to improve on the baseline benchmark performance, while the modified regression trained on data excluded the outliers outperformed all the other models including the baseline benchmark.

racy.

8. Conclusion

Execution time prediction is a notoriously difficult problem for any modern processor, but is especially difficult for processors using the expansive x86-64 instruction set. We achieved our initial goal of improving upon the baseline while maintaining necessary throughput, lowering the average error rate by 16% and the threshold error rate by 29%. However, it was frustrating that all attempts at feature generation for modeling the behavior of outliers performed so poorly. Using features to represent specific opcodes over certain register-pairs, we were able to model some common causes of large execution time, but the inability of those features to predict consistently had a net negative effect on the model's overall performance. In addition, the unmodeled effects essentially impose an upper-bound on any predictor without deeper information about processor state during execution. Despite the difficulties modeling the outliers in our data, the improvement in error rates achieves our goal of developing a better performance model for loop-free x86 code sequences while maintaining the required prediction throughput.

References

[1] Eric Schkufza, Rahul Sharma & Alex Aiken - Stochastic Superoptimization. Stanford University

[2] Agner Fog. Technical University of Denmark. (2013) Instruction tables Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. http:www.agner.orgoptimizeinstruction_tables.pdf (12102013)