Predicting x86 Program Runtime for Intel Processor

Behram Mistree, Hamidreza Hakim Javadi, Omid Mashayekhi
Stanford University
bmistree,hrhakim,omidm@stanford.edu

1 Introduction

Programs can be equivalent: given the same set of inputs, they always produce the same set of outputs. Superoptimizing compilers leverage this notion of equivalence, substituting one program (or section of program) with an equivalent program that runs faster. Recent work by Schkufza attempts to build a superoptimizing compiler that intelligently searches a program space composed of all equivalent sets of instructions [3]. This work relies on being able to predict a program's runtime given its program text.

This report explores several runtime prediction models. Specifically, given a set of loop-free x86 input instructions, our project predicts how long a processor takes to execute those instructions.

Section 2 discusses predicting runtimes for fixed length programs and shows that kernelized regression models can predict program runtime with 12% average error compared to a gold-standard tool [1]. Following these results, Section 3 proposes and evaluates a generative model for predicting runtimes for programs with arbitrary numbers of instructions. Finally, Section 4 provides a performance analysis of the proposed predicter.

2 Fixed length programs

This section builds models to predict runtimes for programs composed of a fixed numbers of instructions. Section 2.1 shows the performance of a simple linear regression and 2.2 expand this model using Gaussian and polynomial kernels.

2.1 Linear regression

Figure 1 shows the results of applying linear regression on randomly generated programs of length 10, 20, 30, and 40 [1]. Each program feature set is a vector containing the amount of time that each instruction would otherwise take to run by itself. These times range from 1 to 47 cycles. Feature weights are learned from a training set of 10,000 programs, each, and results are plotted against a validation set of 2,000 programs each. Normalized error is calculated following:

$$NormalizedError = \frac{|pred - iaca|}{iaca} \quad (1)$$

[1] Section 4.1 extends this to real programs disassembled from objdump.

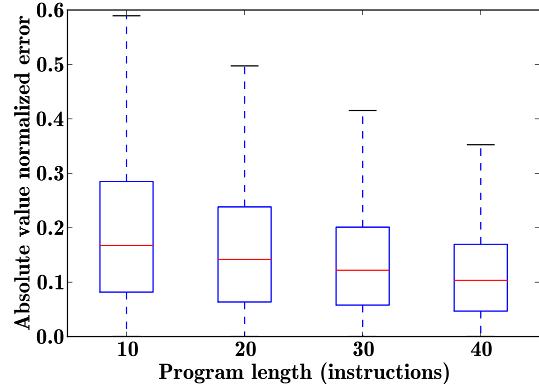


Figure 1: Normalized error for randomly generated programs using linear regression across programs of different lengths.

where *iaca* is the number of cycles a gold standard, proprietary tool [1] predicts as program runtime.

By itself, linear regression performs well: across all program lengths tested, linear regression produces an average normalized error of only 0.158.

The results in Figure 1 suggest an interesting trend: as programs increase in length, a basic linear regression works better and better. In some ways, this is surprising. Althgouh the training set for each experiment shown in Figure 1 is the same size, the size of the space of potential programs that each model tries to learn is very different: the space of programs of length 20 is roughly 1500^{10} larger than the space of programs of length 10 (there are approximately 1500 separate x86 instructions).

This means that for increasing program lengths, each regression learns over a training set that is more likely to be less and less representative of what it is trying to predict, i.e., for increasing program lengths, we should expect more error from *variance*.

Table 1 plots normalized average error for programs of length 10, 20, 30, and 40 as training set size increases. Note that after 1000 samples, increases in training set size have almost no impact on test set error. This suggests that the generalization error shown in Figure 1 is less a matter of under-representativeness of the training set and potentially more a matter of bias in the underlying regression.

Prog Len	$\|S_{tr}\| = 100$	500	1k	2k	5k	10k	15k
10	.235	.215	.204	.207	.203	.200	.200
20	.176	.170	.168	.161	.165	.166	.164
30	.158	.138	.144	.143	.139	.143	.138
40	.159	.133	.132	.125	.131	.124	.124

Table 1: Normalized error as training set size increases.

2.2 Kernelized regression

The previous subsection argued that error from linear regression was likely attributable to model bias. Kernelizing the regression can reduce model bias: whereas linear regression only captures linear dependencies between the features and the output of the model, kernelizing can capture more complex relationships.

We use the Kernel trick [6], replacing each original feature vector, $x^{(i)}$, with a (potentially) high-dimensional feature vector, $\Phi(x^{(i)})$ containing nonlinear functions of the input features, and then we solve linear regression using these high dimensional vectors as our new feature vectors. Therefore, we solve:

$$\min_{\theta} \quad \sum_{i=1}^{m} (\theta^\top \Phi(x^{(i)}) - y^{(i)})^2$$

One issue in doing regression (especially nonlinear regressions) is that the solution to the regression problem can be a very complex and non-smoothe function, which can cause high variance. In order to overcome this problem, we use Tikhonov regularization [5]. Specically, we add $\lambda||\theta||_2^2$ to our cost function, which penalizes non-smoothe predicted models. (The intuition behind this is that if our predicted model has smaller coefficients then the model will be more smooth and we will have less variance). Therefore, we solve the following problem:

$$\min_{\theta} \quad \sum_{i=1}^{m} (\theta^\top \Phi(x^{(i)}) - y^{(i)})^2 + \lambda||\theta||_2^2$$

Using the Kernel trick and the Representer Theorem [4], we can show that the

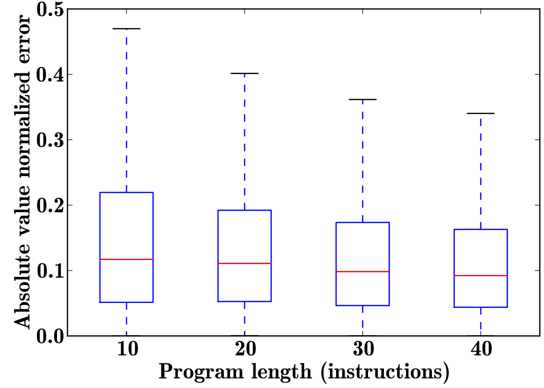
$$y^{predicted} = \theta^{*\top} \Phi(x^{(i)}) = \sum_{i=1}^{m} c_i K(x^{(i)}, x^{(new)})$$

where:

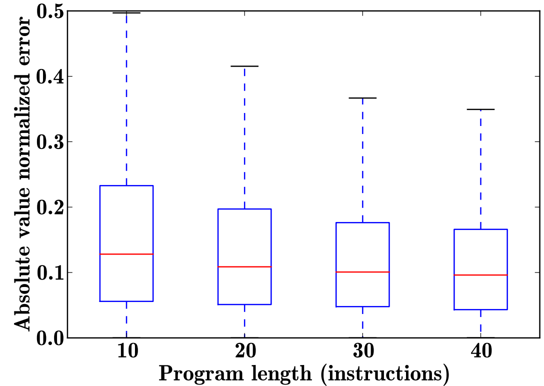
$$c = (K + \lambda I_m)^{-1} y^\top$$

Using these results, we expanded our linear regression to use:

1. a Gaussian kernel with parameter σ (bandwidth) in which $k(x^{(i)}, x^{(j)}) = e^{-\frac{||x^{(i)} - x^{(j)}||_2^2}{2\sigma^2}}$ and
2. a polynomial kernel with parameters d(degree) and c in which $k(x^{(i)}, x^{(j)}) = ((x^{(i)})^T x^{(j)} + c)^d$.



(a) Gaussian kernel regression



(b) Polynomial kernel regression

Figure 2: Normalized error for Gaussian and polynomial kernels on fixed size programs.

Figure 2 presents the results of the Gaussian (σ^2=10) and polynomial kernelized regresssion (d=2, c=1). Both approaches showed only mild improvements compared to linear regression: on average, 3% for the Gaussian kernel and 2% for the polynomial.

There are several possible reasons that these kernels only show modest improvements. Section 3.2 explores one of the most basic of these: changing the mapping from x86 program text to features.

3 Variable length programs

The previous section discussed predicting runtimes for a set of programs of fixed length. However, programs of interest can have varying numbers of instructions: a superoptimizing compiler is primarily concerned with the question *What is the fastest equivalent program of any number of instructions?* not *What is the fastest equivalent program of ten instructions?* A patchwork approach that trains separate models over the entire space of program lengths following the methodology outlined in Section 2.1 is feasible for our target application. However, from a machine learning standpoint,

```
sub_program_vec = [];
runtime_estimate = 0
instruction inst;

for inst = program.first_ins until
    inst = prgram.last_inst:
    runtime_estimate +=
      increment_time(sub_program_vec, inst)
    sub_program_vec.append(inst)
end
```

Figure 3: Pseudo code for generative runtime prediction algorithm.

it is relatively uninteresting.

This section instead explores a different approach. It attempts to build a *generative model* of program runtime. Instead of predicting the runtime of a fixed length program, the generative model estimates the length of a subset of a program and iteratively updates this estimate as additional instructions are added to the end of the program. Figure 3 shows high-level pseudocode for this algorithm.

In essence, a generative model predicts the incremental change a single additional instruction makes to program runtime. A good generative model therefore addresses the challenge of programs of different lengths: one can predict the runtime of an arbitrarily-length program by progressively updating a runtime estimate for each instruction in the program.

Section 3.1 presents a naive generative model. Section 3.2 analyzes the deficiencies of this naive model and uses them to motivate more complicated feature selection discussed. Section 3.3 uses these features to build more complicated generative models and discusses their performance.

3.1 Naive generative model

This section describes and shows results on programs from implementing the simplest possible generative model, which we call the *naive generative model*. The naive generative model decomposes a program into its individual component instructions. It then assigns each instruction a weight that is the (precomputed) average of the time it takes to run that instruction on all sets of valid inputs (e.g., registers and immediates). Finally, the naive generative model calculates the overall program runtime as the sum of each of its instructions weights.

Figure 4 shows the performance of this model for 1000 programs composed of a single instruction, four instructions, eight instructions, and sixteen instructions, each. The vertical axis of each subfigure shows each program's runtime (in cycles) predicted from a gold standard proprietary tool [1] and the horizontal axis shows the runtime predicted by the naive generative model. Each subfigure shows the 45° line corresponding to optimal predictions: any point on this line is a correct prediction by the naive generative model of program runtime.

Note that although each subfigure presents data over 1000 programs, fewer than 1000 points are distinguishable because many points overlap. This is particularly true for programs composed of single instructions, which can achieve fewer predicted outcomes than larger programs composed of many instructions. A least squares line (plotted in green) gives a sense of data spread for overlapping data.

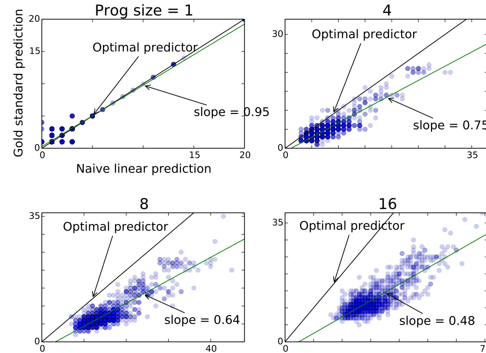


Figure 4: Naive generative prediction error. The horizontal axis shows the number of cycles that the naive generative model predicts; the vertical axis shows the number of cycles predicted using the gold standard IACA tool [1]. An optimal predictor would (predicted cycles equals gold standard cycles) is labeled on each graph as a black line. The green line shows the results of fitting a generative regression through the data.

As can be seen in Figure 4, as program size increases, the naive generative model performs poorly: average error for single instruction programs is almost zero (as indicated by how closely the generative regression line matches the optimal line); while the naive predictor for sixteen instruction programs, on average, has a misprediction error of almost 100%. This trend is to be expected. Implicilty, the naive generative model essentially trains across the space of single instruction programs. Therefore, when it is tested on a single instruction program, it predicts well. However, when it is tested on programs with more instructions (that therefore were not in its training set), it performs poorly.

The likely cause for this error is that the naive generative model ignores processors' potential parallelization and pipelining. Training a model to independently predict runtime of individual instructions and summing the prediction for each does not capture this feature of processors. The following section explores mappings that preserve register read-write conflicts.

3.2 Feature mapping

The degree that a program can be parallelized partially depends on read-write dependencies on registers between instructions. For instance, consider the two, short programs shown in Figure 5. The semantics of each are simple. In the

```
Program 1:
  movl $0x40, %rax
  addl $0x40, %rbx, %rbx

Program 2:
  movl $0x40, %rax
  addl $0x40, %rax, %rax
```

Figure 5: Read-write dependencies

first, the `movl` instruction sets the value of register `rax` to a constant (0x40); the `addl` instruction adds a constant (0x40) to the contents of register `rbx` and puts this result in `rbx`. The second program is similar, except the `addl` instruction is over `rax`. In the first program, the processor can run both the `movl` and `addl` instructions in parallel. However, in the second, the `addl` instruction explicitly reads from a register (`rax`) the `movl` instruction writes to, and therefore the processor cannot run both in parallel. Therefore, although composed of identical instructions, Program 2 takes longer than Program 1 to run.

Both of the feature vectors described in the experiments of Sections 3.1 and 2.1 did not track register dependencies: for both Programs 1 and 2, they produce identical feature vectors.

To improve results for the generative algorithm, we updated our feature mapping to include read-write dependencies. For each instruction in a program, we parse the instruction's name as well as each of the instruction's register arguments. Importantly, registers with different names can refer to parts of the same physical register [2]. For instance, the name `al` refers to the lower order bits of the same physical register that the name `eax` refers to. We track this aliasing as well as which portions of a register can be operated on concurrently.

Given an instruction's name, we index into a pre-compiled database that returns

1. A list of *implicit registers* — registers not specified by the arguments to the instruction — that the instruction reads from and writes to[2].

2. A list of which of its arguments it reads from and writes to.

For instance, the database may specify that the instruction

```
movx a,b
```

reads from its first argument (a) and writes to its second argument (b), but does not operate on any implicit registers.

Following parsing and this database lookup, for every line in a program, we can define its full register read and write set. From these read and write sets, we can easily infer register conflicts between a program's instructions.

We have built a feature mapping function, ϕ, parameterized by a bandwidth parameter τ, that models these read-write dependencies. More concretely, define a program, P, as the ordered set $I_1, I_2, I_3, ..., I_m$, where I_i is the ith instruction in P. Then the feature vector associated with appending the instruction I_{m+1} to this program is

$$\phi(I_{m+1}, \tau, P) = \begin{bmatrix} C(I_{m+1-\tau}) \times \mathbf{1}\{I_{m+1-\tau} \rightarrow I_{m+1}\} \\ C(I_{m+2-\tau}) \times \mathbf{1}\{I_{m+2-\tau} \rightarrow I_{m+1}\} \\ \vdots \\ C(I_m) \times \mathbf{1}\{I_i \rightarrow I_{m+1}\} \\ C(I_{m+1}) \end{bmatrix}$$

where, $C(.)$ is the number of cycles it takes to run a program, and $I_j \rightarrow I_k$ is true if the jth instruction of P writes to a register that the kth instruction of P reads or writes to. In this model, the bandwidth parameter, τ, specifies how many instructions to look back from the current instruction for read-write conflicts. Section 4.2 discusses the performance of this feature creation approach, as any proposed solution should be fast enough to be useful in practice.

3.3 Generative model

In order to predict the incremental runtime that each instruction adds to the total program runtime, we have devised the following learning process. For programs of length 10, take each instruction, say I, and collect the features as explained in Section 3.2. Then run the program with and without I, and use linear regression to learn the incremental runtime that I adds to the program from the collected features.

For a given program with arbitrary length L, we add the predicted incremental runtime of each instruction based on the collected features to predict the total runtime. Figure 6 shows the results. As the bandwidth increases the predictions becomes more accurate. However, the difference for bandwidths greater than 5 is insignificant. Note that as we saw in Section 2.2, the Gaussian or polynomial kernel are almost the same as linear regression so we only show the result for linear regression, here.

4 Conclusions and extensions

This report examined using linear and kernelized regressions to predict program runtime for x86 programs. For programs of fixed length we were able to predict program runtime with 12% average error compared to a gold-standard tool. For variable lengthed programs, our generative model performs similarly, and is flexible enough to predict programs of arbitrary size. The following subsections discuss additional experiments we ran using our model.

[2] Instructions can also "zero-extend" registers, which writes zeros across the higher order bits of a physical register.

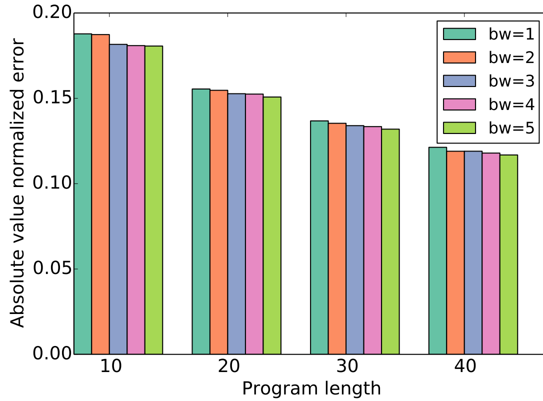


Figure 6: Normalized error of generative model with different bandwidth parameter. Training set size is composed of 800 programs with length 10. But, each test set has 200 programs of length 10, 20, 30, and 40.

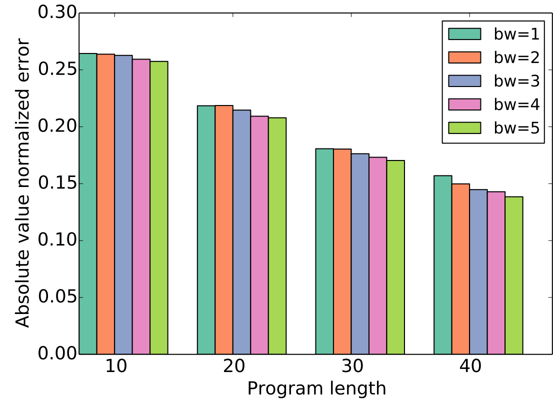


Figure 7: Normalized error of generative model for disassembly of gcc using linear regression with a training set size 800 of and test set size of 200.

4.1 Real programs

The dataset introduced in Section 2.1 and used throughout this report was composed of random sequences of instructions. This random dataset matched the project sponsor's goals — to intelligently search the space of *all* x86 programs. However, we were curious how well our model predicted runtimes of compiler-generated programs, which have more regular structure.

To explore this question, we ran a disassembler (`objdump`) across programs compiled using gcc. Because the model produced in the previous sections was for loop-free programs, we filtered jump instructions from the disassembled code. This filtering limits our ability to draw strong conclusions about the model's accuracy for real programs, but likely still produces more realistic code segments than the previous randomly-generated dataset.

Figure 7 shows the normalized error from running the generative model with linear regression on a disassembly of the gcc binary. Running the same regressions on other program types (e.g., programs that make heavy use of IO, gui-based programs, etc.) produced similar results. Average normalized error is approximately 5% higher for this real data.

4.2 Feature collection runtime

Our collaborator's target application requires not only predicting runtime accurately, but *quickly* as well. To ensure that the feature collection described in Section 3.2 does not become a prediction bottleneck, we benchmarked the performance of our unoptimized Python feature collector. Table2 shows these numbers collected on a 2.80GHz processor. With a bandwidth parameter of 5, our unoptimized Python feature collector is able to produce features at a rate of approximately 100k/second. Coupling these numbers with the time required to perform linear regression, the generative model described in Section 3.3 produces predictions orders of magnitude faster than issuing a system call to execute the gold standard tool (6500 predictions/s for generative linear regression model on programs of 10 instructions vs 22 predictions/s for the gold-standard tool).

	bw=1	**2**	**3**	**4**	**5**
Instr/sec	154,502	133,610	119381	105,970	97,308

Table 2: Feature vector construction time versus bandwidth.

References

[1] Intel Architecture Code Analzyer. `http://software.intel.com/en-us/articles/intel-architecture-code-analyzer`.

[2] x86 Structure. `http://en.wikipedia.org/wiki/X86#Structure`.

[3] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 305–316. ACM, 2013.

[4] B. Schölkopf, R. Herbrich, and A. J. Smola. A generalized representer theorem. In *Computational learning theory*, pages 416–426. Springer, 2001.

[5] A. N. Tikhonov and V. Y. Arsenin. Solutions of ill-posed problems. 1977.

[6] V. Vapnik. *The nature of statistical learning theory*. springer, 2000.