Efficient Program Compilation through Machine Learning Techniques

Gennady Pekhimenko[1*] and Angela Demke Brown[2]

[1] IBM, Toronto ON L6G 1C7 , Canada, gennadyp@ca.ibm.com
[2] University of Toronto, M5S 2E4, Canada, demke@cs.toronto.edu

Abstract. The wealth of available compiler optimizations leads to the dual problems of finding the best set of optimizations and the best heuristic parameters to tune each optimization. We describe how machine learning techniques, such as logistic regression, can be used to address these problems. We focus on decreasing the compile time for a static commercial compiler, while preserving the execution time. We show that we can speed up the compile process by at least a factor of two with almost the same generated code quality on the SPEC2000 benchmark suite, and that our logistic classifier achieves the same prediction quality for non-SPEC benchmarks.

1 Introduction

Decades of compiler optimization research have yielded scores of code analyses and transformations, yet this "embarrassment of riches" creates new problems. Applying all available optimizations can lead to unacceptable compile times, even for static compilers. Moreover, it is well known that transformations are not always beneficial, and that there is no single set of optimizations [2, 3, 4], order for applying optimizations [2], or choice of parameter values for the heuristics used in optimizations [5, 6, 7] that will result in the fastest generated code for all programs. How then is the compiler writer to choose which optimization passes to apply as defaults? Similarly, how should an application developer choose the compiler options to apply to a particular program? The solution for this task is still a mixture of art and experience, often requiring a great amount of time and effort by a human expert.

Choosing and tuning optimizations is difficult for two main reasons. First, the highly non-linear interaction between optimizations means that a change in one transformation requires unpredictable adjustments to others. Second, the large number of options makes it impossible to check the search space completely. Machine learning (ML) algorithms are an appealing option for helping to automate this process, and have been applied previously to various aspects of

* © IBM Corporation, Gennady Pekhimenko, and Angela Demke Brown, 2009. This research is done under Pekhimenko's MSc. thesis [1] in 2008 and supported with the Natural Sciences and Engineering Research Council of Canada and the IBM Centre for Advanced Studies.

the problem [5, 3, 4, 8, 6]. Most of these efforts focused on decreasing execution time or total time (in the dynamic case), but for commercial static compilers the compilation time can also be an issue. Decreasing the compilation time, while preserving the quality of the generated code, is our primary goal.

1.1 Motivation

It has traditionally been argued that static compile time is not an issue because compiling is a one-time process, while the optimized generated code is used frequently. This argument does not hold for large, ongoing software projects (e.g., operating systems or databases) that are frequently recompiled during development, testing, and debugging. In many cases, it is necessary to use the same compilation options as the deployed product, for which the most efficient code possible is desired. Today's commercial compilers have a significant number of powerful, but time-consuming, transformations. Lacking a good way to determine when and where each transformation is needed, compiler writers typically apply them all to the whole program[3]. To illustrate the magnitude of the problem, it can take more than one day to compile a commercial operating system using IBM's TPO (Toronto Portable Optimizer). Reducing this delay, while preserving the generated code quality, could improve the development cycle significantly.

Our goal is to use the full power of the optimizer only if it is needed. This strategy may also reduce the execution time by preventing "bad" transformations. Applying the same set of transformations to the whole program is unlikely to give the best results [3, 9], since individual methods have widely varying characteristics. Thus, we aim to select transformations on a per-method basis. To this end, we require a way to (a) predict what transformations are needed for a particular method, and (b) apply only this list of transformations. To differentiate between methods, we must identify their distinguishing characteristics, which are referred to as *features*. Given this description, we can learn a function to predict the best set of transformations for a method with a given set of features.

We chose to apply machine learning to the optimizer component of the IBM® XL compilers, known as TPO (Toronto Portable Optimizer), which is an aggressively tuned commercial product with a large number of optimizations. We show that machine learning can bring benefits that are impractical to achieve manually due to the complexity of this optimizer. We chose to use *logistic regression* for learning because it is a good match for our problem, and has been proven effective for a similar problem in a dynamic compiler [3]. We consider both the problem of choosing which optimizations to enable, and the problem of tuning heuristic parameter values for certain optimizations.

To restrict the problem somewhat, we chose to investigate only the set of transformations that differ between two high levels of optimization in TPO, one

[3] Different optimization levels (e.g., -O2, -O3) can be used to trade off compile time and code quality, but all selected transformations are applied to the whole program.

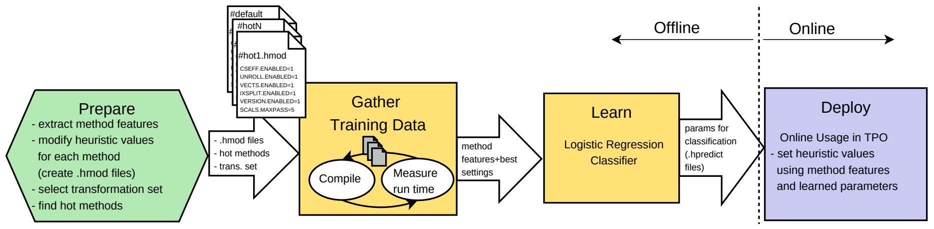

#default
#hotN
#hot1.hmod
CSEFF.ENABLED=1
UNROLL.ENABLED=1
VECTS.ENABLED=1
IXSPLIT.ENABLED=1
VERSION.ENABLED=1
SCALS.MAXPASS=5
Offline
Online
Prepare
- extract method features
- modify heuristic values for each method (create .hmod files)
- select transformation set
- find hot methods
- .hmod files
- hot methods
- trans. set
Gather Training Data
Compile
Measure run time
method features+best settings
Learn
Logistic Regression Classifier
params for classification (.hpredict files)
Deploy
Online Usage in TPO
- set heuristic values using method features and learned parameters

Fig. 1. Overview of our approach

with the desired execution time (*-qhot -O3*) and one with the desired compilation time (*-O3*). There are approximately 50 transformations between these levels. To determine the maximum potential, we measured the compile time and the resulting execution time for 21 of the SPEC2000 [10] benchmarks. The results were promising, showing that *-qhot -O3* reduces execution time by 18% on average (ranging from 2% to 2X) but increases compile time by 3X on average (ranging from 2X to greater than 6X), as compared to *-O3*.

The remainder of this paper is organized as follows. Section 2 discusses our methodology while our experimental setup is described in Section 3. Section 4 presents and discusses the results. Related work is discussed in Section 5. We conclude and describe the potential of this work in Section 6.

2 Design and Implementation

In this section, we describe our approach, which can be separated into four non-intersecting phases: preparing for data collection, gathering training data, learning, and deployment. Figure 1 provides an overview. Because of space limitations, we omit many of the details, which can be found in Pekhimenko's thesis [1].

2.1 Preparing for Data Collection

Before we can generate a set of training data for a machine learning algorithm, we must first have three key elements in place. These are: (1) a means of selecting and calculating the features that are used to characterize a method, (2) a means of modifying the heuristic values on a per-method basis, and (3) a target set of transformations for which data will be collected.

Feature Extraction Features describe characteristics of each method that may be relevant to particular optimizations; if they are chosen well, they can be used to predict the best set of transformations for a new method. A good feature should meet two conditions. First, it should describe a method characteristic that is relevant to some transformation, making it likely that methods that share a similar value for the feature will benefit from the same transformations.

Second, it must be fast to compute, because extracting features adds overhead to every method that is compiled and our goal is to reduce compile time.

The set of features we used can be separated into two major categories: general and loop-based. The general features are the ones that characterize a method as to the number and/or percentage of a certain type of instruction (e.g., the number of loads or the percentage of branches). The second group consists of the features that are either loop characteristics or need the loop tree to be computed. Since their calculation depends on a specific compiler structure (the loop tree), they can be computed only after certain phases in TPO.

We used 29 features to describe a method, mostly reusing the characteristics that were already used by the compiler to guide transformations[4]. We do not claim that this is the best possible set of features, but it is similar to those used in other work [5, 3, 6] and appears sufficient to describe a method. We define a feature vector $\bar{\mathbf{x}}$ that represents each method for our classifier. For example, the *resid* method of the *mcf* SPEC2000 benchmark has the feature vector

$$\bar{\mathbf{x}} = \{\, \mathbf{1379}, 558, 0.4, 20, 0.01, \mathbf{64,}\ \mathbf{0.05}, 10, 0.01, 16, 0.01, 0.01, 222, 0.16, 0.05, \\ \mathbf{2}, 6, 6, 1, 0, 1, 1, 0, 0.33, 1, 0, \mathbf{1}, 33, 227\}$$

Referring to the bolded entries, this means that *resid* has 1379 total operations, 64 float operations (5% of the total number), a 2-level nested loop, and multi-dimensional accesses inside the loops.

All our features are static, so they do not depend on the program input and do not always characterize the method behavior well. Using dynamic characteristics measured by performance counters [4] could be even more powerful.

Implementation: To collect the per-method feature vectors, we created a new C++ class called *FeaturesCollection*. It follows the TPO rules for a single transformation and can be called as a regular optimization in the transformation list. Every feature has a member in the *FeaturesCollection* class (e.g., *int32 opersNmbr* for the total number of operations). Feature computation is separated into two methods. The first method calculates the general features that do not need the loop tree. The second method calculates features that are "loop tree" dependent (e.g., the maximum nest level or the presence of multi-dimensional array accesses in the loops). The whole class implementation is about 1000 lines of C++ code and is separate from the rest of the TPO code.

Modifying Heuristic Values For most current compilers, default heuristic values are embedded within the transformation code. This makes it hard to control multiple heuristic properties from a central place. Adding new functionality to the compiler can be quite cumbersome and error-prone, because of the need to find and update every heuristic property that is affected by the change. An alternative is to record all of the heuristic values in a central data structure, which can be read, updated, and dumped to a file quickly.

[4] A complete list of the features used can be found in Pekhimenko's thesis [1].

```
CSEFF.ENABLED=0
UNROLL.ENABLED=0
BALTREES.ENABLED=0
VECTS.ENABLED=0
IXSPLIT.ENABLED=0
VERSION.ENABLED=0
WANDWAVING.ENABLED=0
SCALS.MAXPASS=2
```

Fig. 2. Hmod file example with heuristic properties

TPO provides this alternative using the Heuristic Context and Heuristic Context Modifiers [11]; similar functionality is available for GCC through the Interactive Compilation Interface [12]. We briefly summarize TPO's Heuristic Context approach here. Each transformation is assigned an abbreviation – a unique text string that will represent it (e.g. the loop unrolling transformation is "unroll"). For each transformation, the set of heuristic properties that control its behavior is also defined, and an abbreviation of each property is assigned. In addition to these two abbreviations (which we will refer to as "transabbrev" and "abbrev"), each instance of a heuristic property has a default value, a range of legal values (e.g., {0,1} for a Boolean type property, or {1-6, 10-20} for an integer type property) and a short textual description. Heuristic Property instances can be accessed by a key composed of "transabbrev"."abbrev" (e.g. "unroll.enabled" allows us to enable or disable loop unrolling). To modify the Heuristic Context and its properties, we can use a Heuristic Context Modifier. For example, to consider register pressure while unrolling loops, we must specify "unroll.regpr=1".

For our purposes, we need to have fine-grained control over heuristic values, so that different values can be chosen for individual methods. A compiler user may also want such a feature, for instance to enable additional transformations for methods that are known to be hot. To provide this flexibility, we added the notion of a *mode* to TPO. A mode specifies a set of heuristic property values, as well as a set of methods to which the mode should apply. We include a default mode for methods that do not have any special mode.

With a compiler option, we can define the available modes and the set of functions for each mode. For each mode, we must also create a file called *mode_name.hmod* in which the heuristic property values are specified, as shown in Figure 2. The default mode must also have a ".hmod" file, called *default.hmod.* We describe how these modes are used to collect training data in Section 2.2.

Choosing the Transformation Set The final step before we can collect training data is choosing which transformations to search through. Every additional transformation increases the search space, but at the same time, ignoring a valuable transformation could reduce the quality of our results. We are interested in those transformations that are (i) included at the *-qhot -O3* level, (ii) not included at the *-O3* level, (iii) take a significant amount of time to compile, and (iv) influence the performance of the generated code.

With the help of the TPO compiler writers, we formed a group of transformations that fit our criteria. Many of the heuristic properties for these transformations are binary (e.g., enabled or disabled); some apply to a set of transformations, rather than a single one. There are a total of 24 heuristic values that we are trying to predict. This gives at least 2^{24} different variants if we assume that every property is binary. Of course, some properties are not binary, for example *SCALS.MAXPASS*, which defines how many times we will apply the set of scalar optimizations, usually takes values from 2 (for *-O3* level) up to 5 (for *-qhot -O3* level). Moreover, this gives only the variants for a single method. For the case where we have N methods for a test, we will get 2^{24*N} as a lower bound on the number of variants. Clearly, it is not possible to try every one.

2.2 Gathering Training Data

For effective learning, it is important to provide examples of "correct" behavior. In our case, we must obtain examples of the best heuristic values to use for a method with a given feature vector. Since we do not know these values *a priori*, we must search for them by trying different settings and measuring the resulting compile and execution times. One of the main problems is how to make an efficient search over a search space that increases exponentially with both the number of heuristic values and the number of methods. In addition to restricting the set of transformations that we consider, we further reduce the search space by only considering the hot methods in each benchmark, as identified by the *tprof* profiling tool. Even with these restrictions, the search space is still enormous.

Other works [3, 5] proposed either a full search for small spaces or the use of randomly generated heuristic values. Full search is clearly not suitable for our problem because of the search space size. Several attempts to gather training data with the random approach did not give any significant decrease in the compile time. However, the characteristics of the problem that we are trying to solve allows us to use another search technique.

The TPO optimizer applies transformations in a fixed order, such that the next transformation uses the output from the previous one as its input. This property suggests the use of an orthogonal search, with two options: (1) start at the *-O3* level and enable new transformations, or (2) start at the *-qhot -O3* level and disable transformations. We decided against enabling new transformations starting at the *-O3* level. Some transformations may depend on steps performed by earlier ones, which do not themselves produce any significant speedup. In this case, it would appear that the earlier transformation was not useful and we would disable it. The later transformation would then appear to be unneeded as well, since it would not be effective without the earlier (disabled) transformation. Instead, we can disable transformations starting with the full set at the *-qhot -O3* level and control that the execution time does not increase significantly and the compile time gradually decreases too. If we start disabling transformations backwards (from the last to the first) we are less likely to break a useful sequence of transformations that gives a speedup.

We do not claim that this approach always gives the best values for heuristics. For example, it is possible that an early transformation X has a negative effect, which is corrected by a later transformation Y. In our approach, we will first consider Y and find that it should be enabled because it improves performance.

```
GenerateTrainingData($tests, hots, trans$)
1 for each $test$ in $tests$ do
2         Init($curr\_settings[test], best\_settings[test]$)
3         for $i \leftarrow length[trans]$ downto 1 do
4                 for each $method$ in $hots[test]$ do
                          ▷ Disable current transformation for $method$ in $test$
5                         $curr\_settings[test][method][i].enabled = 0$
                          ▷ Run $test$ and get the compile and execution time
6                         $result = Run(test, curr\_settings, \&curr\_comp, \&curr\_exec)$
7                         if IsBetter($result, curr\_comp, curr\_exec$)
8                            then Update($best\_settings[test], curr\_comp, curr\_exec$)
9                            else $curr\_settings[test][method][i].enabled = 1$
```

Fig. 3. The algorithm for Generating Training Data

Working backwards, we will later find that X is not useful and disable it. At the end, we will have transformation Y enabled, but it is no longer necessary after X has been disabled. Another potential problem is that orthogonal search can be sensitive to the order in which transformations are tested [7]. We assume that the compiler writers chose a good initial order. In spite of these issues, this approach does allow us to find the best (or nearly the best) set of transformations that linearly depends on the number of heuristic parameters, which makes it practical.

Figure 3 presents pseudo-code for generating training data. For simplicity, we present the algorithm as though all heuristic values are binary; however, our implementation also considers non-binary values. For every test we have a set of hot methods obtained by using the *tprof* tool. Using the Heuristic Context Modifiers mechanism, we can set heuristic values, enabling or disabling transformations and trying different values for non-binary properties. Each hot method uses a different ".hmod" file, allowing us to set the heuristic values for each one independently; the "default.hmod" file is used for all other methods. After the execution of the current test, we have results: the test correctness, the compile time, and the execution time.

Next, we check whether the current result is better than the previous best result (line 7), which requires two checks. First, we check that the new execution time is not significantly (e.g., not 1%) worse than the best execution time and the baseline execution time at the *-qhot -O3* level. The same checks are performed for the compile time. The baseline check is needed to avoid a gradual degradation. For example, a 1% increase in every execution during 200 runs can cause a 3 times increase in the execution time. Normally, the compile time should always decrease, but there can be two cases when this does not happen. The first case is a simple variation that results from the limited accuracy of the time measurement (this may happen even between two runs of the same executable). The second case occurs when disabling a transformation causes others to perform much more work and, hence, increases the compile time.

Finally, we save the best heuristic values that we found together with the feature vector for each method to be used in the learning process.

2.3 Learning Process

As with the training phase, the learning phase in our approach can be done offline. Thus, learning does not impose any overhead at compile time. Instead, we simply use the learned parameters together with the features (which are easy to compute) to predict the best set of transformations for a new function. This makes it possible to use any learning algorithm we want with the same straightforward deployment implementation.

After gathering the training data for every method, we have a feature vector $\bar{\mathbf{x}}$ and the corresponding vector of the best transformation set $\bar{\mathbf{C}}$. Our goal is now to find, or at least approximate, the function: $F(\bar{\mathbf{x}}) = \bar{\mathbf{C}}$. We can then apply this function to a new feature vector $\bar{\mathbf{x}}^{'}$, and approximate its best transformation set $\bar{\mathbf{C}}^{'}$ with some good $\bar{\mathbf{C}}^{''}$.

Our problem is the classical ML problem and several powerful methods were invented to solve it, i.e., nearest neighbors and artificial neural networks.

We used logistic regression with penalty regularization as an easy and effective means of classification. We have a 29 dimensional space of features and a 24 dimensional space of outputs that can be considered as 24 single outputs for simplicity; and we do not have any prior knowledge (such as sparsity or features' dependencies) about our training data to apply something special.

Logistic Regression Logistic Regression is a popular linear classification method. Its predictor function consists of a transformed linear combination of explanatory variables. The logistic regression model consists of a multinomial random variable y, a feature vector $\bar{\mathbf{x}}$, and a weight vector θ. In our case, y is the possible value of the heuristic property. We initially set the weight vector θ with some random values; later, they will be changed with the ones that maximize the *conditional log-likelihood* (2). The posterior of y is the "softmax" of linear functions of the feature vector $\bar{\mathbf{x}}$, as shown in Equation 1.

$$p(y = k|\bar{\mathbf{x}}, \theta) = \frac{\exp(\theta_k^\top \bar{\mathbf{x}})}{\sum_j \exp(\theta_j^\top \bar{\mathbf{x}})} \tag{1}$$

To fit this model, we need to optimize the *conditional log-likelihood* $\ell(\theta; D)$:

$$\ell(\theta; D) = \sum_n \log p(y = y^n|\bar{\mathbf{x}}^n, \theta) = \sum_{nk} y_k^n \log p_k^n \tag{2}$$

where $y_k^n \equiv [y^n == k], p_k^n \equiv p(y = k|\bar{\mathbf{x}}^n)$.

Implementation Our current classifier was implemented in Matlab and consists of several *.m* files that allow us to calculate likelihood and its gradient, extract data for different data sets, and perform minimization. We use a set of Perl scripts to (1) collect the baseline compile and execution times for the *-O3* and *-qhot -O3* levels, (2) collect method features for training, (3) gather training data by executing benchmarks and collecting results, and (4) transform the training data into a form that can be used in Matlab for classification.

The learning process generates a collection of files (called *hpredict* files), one for each heuristic considered (e.g., *loops.txt*, *unroll.txt*, etc.). Every file contains the vector of learned parameters, θ, produced by the classifier.

2.4 Deployment Process

After completing the learning process, we have a vector of optimum parameters θ that we can use to predict the transformation set using the feature vector $\bar{\mathbf{x}}$. To achieve this, we just need to use Equation 1 for every transformation. If the probability is greater than 0.5, then we will apply the transformation; otherwise, we will not apply it. Some researchers have required a higher probability before applying a transformation (e.g., $p > 0.6$ [3]); we have not investigated other threshold values.

During compilation, TPO loads the *hpredict* files generated offline by the classifier. The *FeaturesCollection* class that we added to TPO (described in Section 2.1) also provides functionality to calculate a single transformation prediction that can be computed using the parameters from the logistic regression. The *CountPrediction* method gives an answer to the question of whether a transformation should be applied to the current method. Using this function for every interesting transformation, we generate the Heuristic Context that will be used for the current method.

3 Experimental Setup

In our experiments we used benchmarks from SPEC2000 [10], and some additional float benchmarks from IBM customers. For training purposes we used twenty-one benchmarks from SPEC2000. This gave us 140 hot (from the top of the *tprof* tool output) functions that were used to collect features and search for the optimal set of transformations.

Our target platform was IBM® PowerPC®. Our evaluation used an IBM server with 4 x IBM Power5$^{\text{TM}}$ 1900 MHz processors and 32GB of memory, running the IBM AIX® 5.3 operating system. We used -02 optimizations in all experiments to compile TPO.

Even on a server without any workload, an application's execution time may vary by a small amount. Since we measure and compare the execution and compile time when generating training data, we need to keep this variation in mind. To solve the problem, we use a small interval around the measured times when performing comparisons. If the intervals intersect, they can be considered as the same time for our purposes. Such a comparison may lead us to miss a small performance degradation. To avoid the accumulation of many small degradations, we compare every new time with the previous best and the baseline, as described in Section 2.2, so it is impossible to become significantly worse than the baseline. Currently we use 1% for the possible fluctuation for compile time and 0.5% for the execution time; these values were found to work well in our experiments.

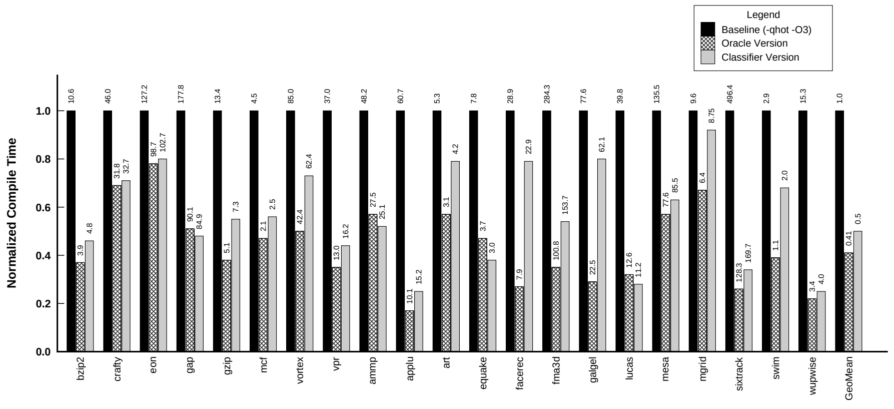


Fig. 4. Compile time comparison between the baseline, the oracle, and the classifier.

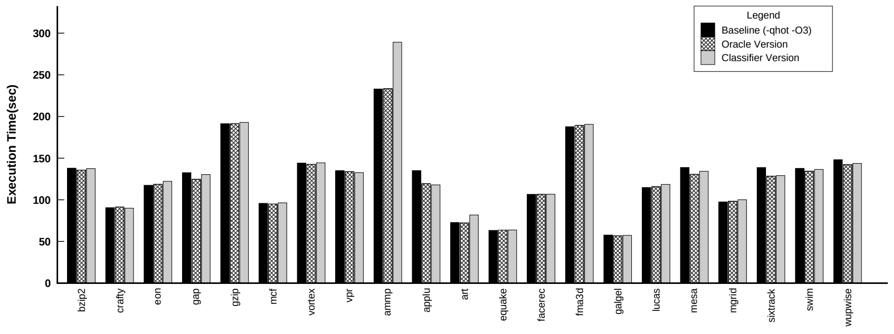


Fig. 5. Execution time comparison between the baseline, the oracle, and the classifier.

4 Results and Discussion

We evaluate two parts of our approach here. First, we assess the effectiveness of our search technique in finding optimal parameter values for the training data. Second, we assess the prediction quality of our classifier.

4.1 Quality of Training Data

Once our search has yielded the optimal heuristic values[5], the first question is, how well can we decrease the compile time? It is unrealistic to expect the classifier to show better results than the optimum we found. Hence, if the training data results are not good enough, it does not make sense to evaluate our classifier.

Figure 4 shows the compile time for the set of optimal heuristic values (labeled *Oracle*) and the compile times achieved using our classifier (labeled *Classifier*), normalized to the compile time of the baseline *-qhot -O3* level. Numbers above the bars provide the raw compile times (in seconds). Focusing on the *Oracle* results, we see that a significant speedup can be achieved for every benchmark, ranging from a speedup of 1.3 for *eon* up to a speedup of 6 for *applu*. The

[5] These may well represent a local optimum, not the globally optimal values.

geometrical mean speedup across all tests is 2.46. For comparison, the compile time of the *-O3* level has a mean speedup of 2.99 over *-qhot -O3*, but this is an unachievable ideal since some additional transformations are needed to meet the execution time goals. We now examine the impact on the execution time.

Figure 5 shows the execution times corresponding to the compile times of Figure 4. Focusing on the *Oracle* bars, we see that using our optimal heuristic values does not cause any significant increase in the execution time. In fact, we actually *decrease* the execution time for six tests (*gap*, *applu*, *mesa*, *sixtrack*, *swim*, and *wupwise*) by preventing harmful transformations. The overall result is a speedup of *1.02*, which is comparable to the results from other works ([5, 6]) where reducing execution time was the primary goal. We emphasize that these results were obtained on a well-tuned commercial compiler, in which significant effort has been made to achieve good performance on the SPEC benchmarks.

One of the most serious drawbacks of our training data is that each transformation is needed in only a few cases, usually for less than 5% percent of the hot functions. These rare cases when a transformation is necessary thus look like noise, which reduces both the stability and the accuracy of our classifier. We expect a more serious set of training data will be needed.

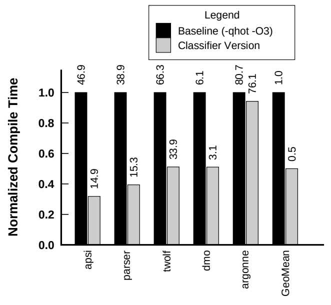
The results of these runs were used to train the logistic regression classifier, which was then incorporated into TPO.

4.2 Classifier Evaluation

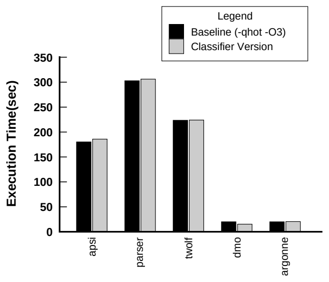
Our main goal was to predict whether a transformation was needed for a given function. The training data gives us this answer, but we consider it only as an *Oracle*, because we cannot afford to collect this information online during compilation. Hence, we need to apply our logistic classifier and evaluate its prediction quality. We performed both an offline and an online evaluation.

Offline evaluation Offline evaluation allows us to measure how many errors our classifier will have on the test set by comparing the optimal values for each test against the predicted ones. This requires that we know the optimal values for the test set. To meet this requirement, we use the standard “leave-one-out cross validation” (LOOCV) technique, which allows us to use our training data as the test data, excluding each function from the training set when “testing” it. The tests are fair since the learning process never uses any information about the function being tested.

There are two types of errors. False positives occur when we predict that a transformation is needed, but it is not. These errors lead to an unnecessary increase of the compile time. False negatives occur when we predict that a useful transformation is not needed. These errors usually lead to a significant increase in the execution time and should be considered more serious. In our experiments, we mostly considered false negatives. The error rate depends on the heuristic property type and is higher for the complex heuristics. For all heuristic properties it was in the 0% - 4% range. Clearly, the error rate depends on the learning process, which itself depends on the initial value of a vector θ (see Section 2.3).



(a) Compile time



(b) Execution time

Fig. 6. Compile and execution time comparison for new benchmarks.

The LOOCV method is easy to implement – we already have all necessary components to do it – but it has one serious drawback. Knowing the number of prediction errors does not tell us how they affect the execution time.

Online Evaluation With the classifier implemented in TPO we can apply its predictions online and measure the effect on compile time and execution time. For this case, we used the parameters that were learned on the full set of hot methods from all benchmarks. Predictions are made for each method, not just the hot ones. This means that our training and testing sets have a small intersection, which may be unfair. To assess the impact of this overlap, we chose several benchmarks at random, excluded their hot functions from learning, and then evaluated them using the new parameters. The results were the same to within the accuracy of our measurements.

The *Classifier* bars of Figure 4 and Figure 5 show the results we obtained on the SPEC2000 benchmarks. The compile time shows an average speedup of **1.99**. This result is worse than the *Oracle*, but it still delivers a significant improvement. In some cases, we have a compile time that is even smaller than with the *Oracle*, but this is not a positive result – it means that we disabled something that was considered valuable when we generated the training data. We expect that these errors should lead to a significant increase in the execution time, as can be seen for *ammp*. In this benchmark, the execution time increases by 18% because of the classifier mispredictions. Overall, however, we see only a 1% increase in the execution time compared with the baseline (*-qhot -O3*).

For these benchmarks, our classifier successfully decreased compile time by a factor of nearly two with a negligible increase in execution time.

4.3 Other Benchmarks

To assess our classifier for benchmarks that were not used in training, we tested three additional benchmarks from SPEC2000 (*apsi*, *parser*, *twolf*), which were excluded for reasons such as having a very "flat" profile without hot functions, or the need for special options to ensure an exact output match. We also used two benchmarks from IBM customers (*dmo* and *argonne*).

Figure 6 shows that we obtained very similar results for this set of benchmarks. We achieved an average speedup of *2* for the compile time and a speedup of *1.04* for the execution time. The execution time speedup was due to the *dmo* benchmark, showing that our classifier is able to disable one or more harmful transformations. We do not have a significant slowdown on any of these benchmarks, so we can conclude that our technique is effective in decreasing the compile time while preserving the quality of the generated code.

5 Related Work

There has been a great deal of research on automatically tuning compiler optimizations, so it is not possible to review it all here. We focus instead on some relevant representative examples covering search techniques and machine learning for tuning single and multiple heuristics.

Iterative compilation searches for the best sequence of compiler options by repeatedly compiling with different settings and evaluating the effect [2, 9, 7, 13]. This technique must be applied to each new program, rather than learning settings that can later be used for previously unseen programs. Nonetheless, improvements to iterative search could improve the quality of our training data.

Pan and Eigenmann [9] search for the best compiler settings for partitions of a program called *tuning sections*. They exploit the fact that the same methods are invoked many times during program execution to evaluate multiple optimization settings during a single run. The average search time is thereby reduced from over 2 hours to under 6 minutes. Agakov et al. [13] use machine learning techniques to generate predictive models that focus the iterative search process. They evaluated two models, both of which require relatively small amounts of training data, using the UTDSP benchmark on two embedded systems. Using their models, they were able to speed up iterative search by an order of magnitude. The benchmarks used in this work were significantly simpler than SPEC2000 (i.e., only 20-500 lines of code), the search space had fewer transformations than we considered, and it was applied at the whole program level. Because of its simplicity, we expect this technique would not be suitable in our setting, but it would be interesting to see if it could be used in the training step.

Many researchers have applied machine learning to the problem of tuning single heuristics. In one early work, Calder et al. [8] focused on static branch prediction using neural networks and decision trees. Training data was easy to collect for this problem by simply observing the actual branch directions. Stephenson et al. [6] used genetic programming to tune hyperblock formation, data prefetching and register allocation. Significant improvement was achieved for the first two cases, but these optimizations were not greatly tuned before. For register allocation, which was already manually tuned, they achieve a more modest average speedup of 2%. Stephenson and Amarasinghe [5] later used nearest neighbors and support vector machines to predict the unroll factor for different loop nests. Problems with the Open Research Compiler (ORC) [14] limited the number of unroll factors they could consider, but they showed an overall speedup on 19 of the 24 SPEC2000 benchmarks, with a 5% average speedup. We achieve similar speedups when considering multiple optimizations on a highly tuned commercial compiler, even though execution time speedup was not our primary goal.

Research that uses machine learning to tune entire sets of optimizations is closer to our work. Cavazos et al. [3] developed an approach that automatically selects the best optimizations on a per-method basis within a JIT compiler. The authors used logistic regression to derive a model that can determine which optimizations to apply based on the features of a method. They show significant speedup for most of the benchmarks. For a dynamic compiler, this speedup may come from reducing the compiler overhead or from improving the generated code, or both. While we were inspired by this research, we target a heavily tuned static compiler with a much larger set of transformations, and tune non-binary heuristic values in addition to enabling or disabling transformations.

Cavazos et al. [4] used dynamic program features, collected with performance counters, to automatically learn a model for selecting compiler optimizations. Before applying the model to a new program, a few initial runs are needed to collect these features. The authors achieved impressive speedups over the highest optimization level of the PathScale EKOPath [15] open-source static compiler, showing that runtime information can be very powerful in describing a program. Our work differs in 3 ways. First, our goal was to reduce compile time, rather than execution time. Second, we selected and tuned transformations on a per-method basis, rather than for the whole program. Third, we were able to achieve our goals using only static features; dynamic features could bring more benefits.

The MILEPOST GCC project [12] has many characteristics in common with our work, including the ability to select transformations on a per-method basis. Although reducing execution time and code size are their primary goals, they report a 22% reduction in compile time for the *susan_corners* benchmark, while reducing execution time and code size by 16% and 13%, respectively. By focusing on compile time, we have achieved greater reductions of nearly 2X on average.

6 Conclusions and Future Work

Though often neglected, static compile time is a significant issue for developers of large software projects. We addressed this problem using the logistic regression machine learning technique to predict the set of transformations, and the values for their heuristic parameters, that are actually needed. We developed a framework for a heavily tuned commercial optimizer, TPO, which selects transformation settings for each method. Our learned classifier can be highly effective. Experiments showed that we can decrease the average compile time by at least a factor of two with a negligible increase in the execution time (1%). However, in the case of misprediction, the increase can be significant. Overall, the compile time comes close to that of the lower optimization level (*-O3*) while still achieving the performance of the higher level (*-qhot -O3*). In the future, we would like to improve our training data, experiment with the use of dynamic features, and apply our framework with the goal of reducing execution time.

7 Acknowledgments and Trademarks

The authors would like to thank the TPO compiler team at the IBM Canada Lab, especially Arie Tal and Raul Silvera. IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies.

References

[1] Pekhimenko, G.: Machine learning algorithms for choosing compiler heuristics. Master's thesis, University of Toronto (January 2008) http://csng.cs.toronto.edu/publication_files/174/pgen_thesis.pdf.

[2] Cooper, K.D., Subramanian, D., Torczon, L.: Adaptive optimizing compilers for the 21st century. Journal of Supercomputing **23**(1) (2002) 7–22

[3] Cavazos, J., O'Boyle, M.F.P.: Method-specific dynamic compilation using logistic regression. In: Proc. of OOPSLA '06. (Oct. 2006) 229–240

[4] Cavazos, J., Fursin, G., Agakov, F., et. al.: Rapidly selecting good compiler optimizations using performance counters. In: Proc. of the 2007 International Symposium on Code Generation and Optimization (CGO '07). (Mar. 2007) 185–197

[5] Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proc. of the 2005 International Symposium on Code Generation and Optimization (CGO'05). (Mar. 2005) 123 – 134

[6] Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta optimization: improving compiler heuristics with machine learning. In: Proc. of the 2003 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI '03). (June 2003) 77–90

[7] Seymour, K., You, H., Dongarra, J.: A comparison of search heuristics for empirical code optimization. In: Proc. of the 2008 IEEE Intl. Conf. on Cluster Computing (3rd Intl Wkshp on Automatic Perf. Tuning). (2008) 421–429

[8] Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B.: Evidence-based static branch prediction using machine learning. ACM Trans. Program. Lang. Syst. **19**(1) (1997) 188–222

[9] Pan, Z., Eigenmann, R.: Fast, automatic, procedure-level performance tuning. In: Proc. of the 15th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'06). (Sept. 2006) 173–181

[10] Standard Performance Evaluation Corporation: SPEC CPU2000 benchmarks. http://www.spec.org/cpu2000/ (2000)

[11] Tal, A.: Method and system for managing heuristic properties. US Patent Application No. 20070089104 (April 19 2007)

[12] Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M.: MILEPOST GCC: Machine-learning-based research compiler. In: Proc. of the GCC Developers' Summit. (June 2008)

[13] Agakov, F., Bonilla, E., Cavazos, J., et.al.: Using machine learning to focus iterative optimization. In: Proc. of the 4th International Symposium on Code Generation and Optimization (CGO'06). (Mar. 2006) 295–305

[14] ORC: Open Research Compiler for Itanium Processor Family. http://ipf-orc.sourceforge.net/

[15] PathScale: EKOPath Compilers. http://www.pathscale.com/