Method-Specific Dynamic Compilation using Logistic Regression

John Cavazos Michael F.P. O'Boyle

Member of HiPEAC
Institute for Computing Systems Architecture (ICSA), School of Informatics
University of Edinburgh, United Kingdom
{jcavazos,mob}@inf.ed.ac.uk

Abstract

Determining the best set of optimizations to apply to a program has been a long standing problem for compiler writers. To reduce the complexity of this task, existing approaches typically apply the same set of optimizations to all procedures within a program, without regard to their particular structure. This paper develops a new method-specific approach that automatically selects the best optimizations on a per method basis within a dynamic compiler. Our approach uses the machine learning technique of logistic regression to automatically derive a predictive model that determines which optimizations to apply based on the features of a method. This technique is implemented in the Jikes RVM Java JIT compiler. Using this approach we reduce the average total execution time of the SPECjvm98 benchmarks by 29%. When the same heuristic is applied to the DaCapo+ benchmark suite, we obtain an average 33% reduction over the default level O2 setting.

Categories and Subject Descriptors D.3 [*Software*]: Programming languages; D.3.4 [*Programming languages*]: Processors—Compilers, Optimization; I.2.6 [*Artificial intelligence*]: Learning—Induction

General Terms Performance, Experimentation, Languages

Keywords Compiler optimization, Machine learning, Logistic Regression, Java, Jikes RVM

1. Introduction

Selecting the best set of optimizations for a program has been the focus of optimizing compilation research for decades. It has been the long term goal of compiler writers, to develop a sequence of optimization phases which analyze and, where appropriate, transform the program so that execution time is reduced. Determining the best set of optimizations and their ordering, however, is notoriously difficult. In fact, Cooper *et al.* [9, 2] have shown that the best ordering is program dependent and that any sequence that is best for one program is unlikely to be the best for another. What we would like is a mechanism that automatically selects the best optimization sequence for a particular program.

In optimizing compilers, it is standard practice to apply the same set of optimizations phases in the same order on each procedure or method within a program. However, just as the best set of optimizations varies from program to program [2, 18], we show that the best set of optimizations varies within a program, i.e., it is *method-specific*. We would like a scheme that selects the best set of optimizations for individual portions of a program, rather than the same set for a whole program.

This paper develops a new method-specific technique that automatically selects the best set of optimizations for different sections of a program. We develop this technique within the Jikes RVM environment and automatically determine the best optimizations on a *per method* basis. Rather than developing a hand-crafted technique to achieve this, we make use of a basic machine learning technique known as *logistic regression* [4] to automatically determine what optimizations are best for each method. This is achieved by training the technique offline on a set of training data which then automatically learns an optimizing heuristic.

Machine learning based techniques have recently received considerable attention as a means of rapidly developing optimization heuristics [20, 7]. Unfortunately, they have been largely constrained to tuning an individual optimization often with disappointing results. For example, although Stephenson *et al.* [20] were able to construct a register allocation heuristic automatically using machine learning, the heuristic was only able to achieve a modest improvement over the existing *hand-tuned* register allocator heuristic.

To the best of our knowledge this is the first paper to automatically learn an overall compilation strategy for individual portions of a program. This means that our scheme learns which optimizations to apply rather than tuning local heuristics and it does this in a dynamic compilation setting. Furthermore, we show significant performance improvement over an existing well-engineered compilation system. Using our approach, for the maximum optimization O2 setting within Jikes RVM, we reduce the total runtime by 29% on average for the SPECjvm98 benchmarks suite. This increases to 33% on a set of benchmarks including DaCapo, SPECjbb2000, and ipsixql. This result is particularly impressive, when considering that recent work on developing by *hand* predictive models to select optimizations [24] achieves performance improvements of less than 3% on average.

The paper is organized as follows. Section 2 provides motivation as to why method-specific optimizations outperform current approaches. This is followed in section 3 by a description of how standard logistic regression can learn whether or not to apply a particular optimization for a particular method. Section 4 then describes the experimental infrastructure and methodology which is followed in section 5 by a presentation of the results. Section 6 provides an analysis of why logistic regression works. Section 7 gives a brief overview of related work and is followed in Section 8 by some concluding remarks.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

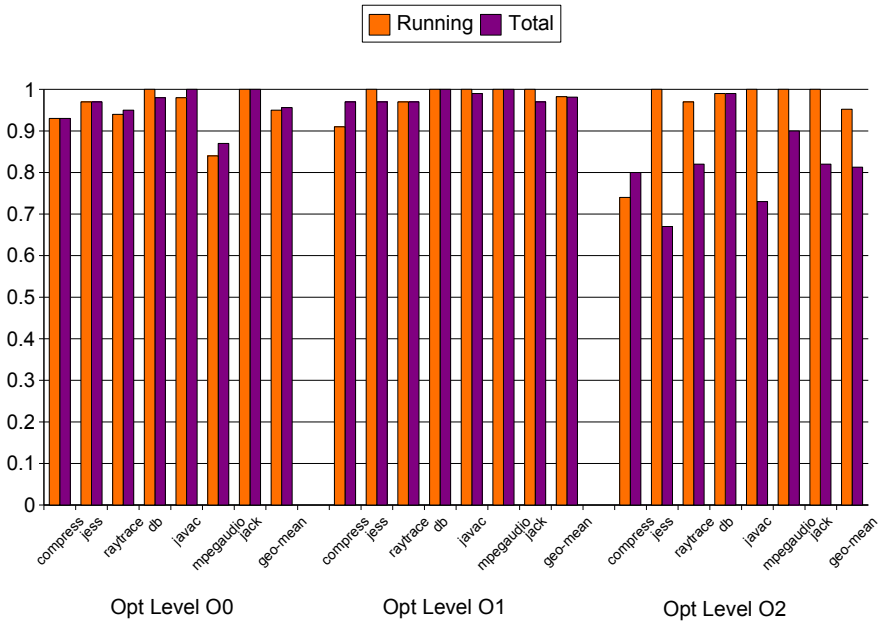


Figure 1. The results of applying the best setting for each method relative to the default setting for each of the different optimization levels. The best setting for each method was found through exhaustive exploration for optimization levels O0 and O1. For optimization level O2, we took the best setting for each method from 1000 random settings evaluated. Total execution time = dynamic compilation time + running time.

2. Motivation

This section shows that selecting the best set of optimizations on a per method basis has the potential to significantly improve the total running time of dynamically compiled programs.

Potential We conducted an initial experiment to determine the extent to which different optimizations affect the total execution time of an application compiled and executed by the Jikes RVM Java JIT compiler. For each method within each SPECjvm98 benchmark, we applied many different optimization settings and recorded the setting that gave the best total execution time on a per method basis. As Jikes RVM has multiple optimization levels, we performed this experiment for the O0, O1, and O2 optimization levels and report the results here.

Due to the way that Jikes RVM is constructed, it is not possible to arbitrarily change the order in which optimizations are applied. Instead, we confine our search to just enabling and disabling optimizations. For optimization levels O0 and O1, which apply 4 and 9 optimizations by default, we tried all possible enumerations of these optimizations, i.e., $2^4 = 16, 2^9 = 512$, respectively. As level O2 applies 20 optimizations an exhaustive enumeration of the 2^{20} settings was not possible, instead we randomly enumerated 1000 different settings and recorded their performance. Table 1 shows the optimizations that are applied by default for these three different optimization levels.

Once the best settings were found for each method, we then dynamically compiled and ran each program using the best optimization settings for each method. Figure 1 presents results of selecting the best set of optimizations at each optimization level on a per method basis for each of the SPECjvm98 benchmarks. As compilation time is dynamic and part of the overall execution time we show two performance results: *running* time and *total* time. Running time indicates the benchmark running times *without* compilation time while total time indicates running times *with* compilation.

The results are plotted relative to the existing default heuristic which is to apply all optimizations grouped at that optimization level (see Table 1).

In the case of optimization level O0, there is on average a 4% reduction in total time available and this reduces to 2% for O1. However, in the case of the highest optimization level O2, there is a significant performance improvement available. Selecting the right optimization for each method gives an average 19% reduction in total execution time.

One size does not fit all Looking at the results, one may conclude that the default settings are poor and that there potentially exists another setting that is not method-specific and would perform well across all other benchmarks.

To test this we enumerated all the different possible combinations of the 4 optimizations which are used by default for optimization level O0 to try to find *one fixed* setting which if applied to *all* methods would provide a performance improvement over the current default.[1] Of the 16 different optimization settings we enumerated, we did not find any single configuration that out-performed the current default. The current default is in fact the optimal fixed setting for the SPECjvm98 benchmarks.

Now, in the previous section we were able to outperform optimization O0 for several of the SPECjvm98 benchmarks giving an average reduction in total execution time of 4%. Thus, an optimization setting tuned for each method is *better* than the optimal fixed setting across the whole program. In other words, for the

[1] Optimization level O0 is an important optimization since it is the first and most often used optimization setting during *adaptive* compilation.

SPECjvm98 benchmarks a method-specific strategy should outperform any program-specific strategy.

Conclusion In our experiments we did not find a single overall optimization setting that significantly improves the performance of the Jikes RVM JIT optimizing compiler. However, selecting a different optimization setting for each method shows the potential to deliver significant gains.

The challenge therefore is to develop a predictive model that can analyze each method and select the optimization set that will reduce total running time. As we are considering dynamic JIT compilation, the overhead of this heuristic must be small otherwise it will outweigh any benefits.

One approach is to handcraft a heuristic based on experimentation and analysis. This is undesirable for two reasons. Firstly, it will be an arduous task and Jikes RVM specific. Secondly, if the platform were to change, the entire tuning of the heuristic would have to be repeated. Instead we use an *machine-learning* based approach which automatically learns a good heuristic. This has the advantage of being a generic technique that easily ports across platforms.

3. Approach

This section gives a detailed overview of how logistic regression based machine learning is used to determine a good optimization heuristic for each of the optimization levels O0, O1, O2, and adaptive within Jikes RVM. The first section outlines the different subactivities that take place when learning and deploying a heuristic This is followed by sections describing how we generate training data, how we extract features from methods, and how these features and training data allow us to learn a heuristic that determines whether or not to apply a set of optimizations. This is followed by an example showing how our learned heuristic is used in practice. Figure 2 outlines our scheme.

3.1 Overview

There are two distinct phases, training and deployment. Training occurs once, off-line, "at the factory" and is equivalent to the time spent by compiler writers designing and implementing their optimization heuristics. Deployment is the act of applying the heuristic at dynamic compilation time. As any learned heuristic incurs dynamic compilation overhead, it is imperative that it be as cheap as possible to evaluate otherwise its runtime benefit will be outweighed by its cost.

Within the training phase, we first generate appropriate training data based on whether we are trying to improve the O0, O1, or O2 optimization levels within Jikes RVM. The training data is randomly generated by applying different optimization settings to each method within each training program and recording their performance on a per method basis using fine-grain timers. We also derive a short description (called a feature vector) of each method so that we can later build a function that takes the feature vector as input and provides the set of optimizations that should be applied as output.

This predictive model once learned is installed into the Jikes RVM compiler and used at runtime as an optimization heuristic. The next sections describe these stages in more detail

3.2 Generating training data

Our aim is to develop a function, **f**, which, given a new method described as feature vector **x**, outputs a vector **c** of 1s and 0s whose elements determine whether or not to apply the corresponding optimization:

$$f(\mathbf{x}) = \mathbf{c}$$

1. Training "at the factory"
   (a) Generate training data results for O0,O1, or O2
      i. Instrument each method with timing calls
      ii. For each method randomly select a set of optimizations and apply
      iii. Record dynamic compilation, running, and total execution time per method
   (b) Generate method features
      i. For each method calculate each element of the feature vector
      ii. Record feature vector for each method
   (c) Learn a model that classifies features
      i. For each method select those optimization settings within 1% of best
      ii. Generate a table where each recorded feature vector is associated with its best optimization settings.
      iii. For each optimization setting **c** determine a probabilistic function **f** that states whether a feature vector **x** should have this optimization set or not.
      iv. Output this set of functions as the learned heuristic
2. Deployment
   (a) Install learned heuristic into Jikes RVM
   (b) For each method dynamically compiled
      i. From bytecodes generate a feature vector
      ii. Use heuristic to determine which optimizations to apply and apply them

Figure 2. Overall technique

Before we can do this, we need to know how different optimization settings affect performance. We therefore try many different optimization settings to see how each setting affects the performance of each method.

For the problem of optimizing Java methods, we can search the entire space of optimizations if the number of optimizations is not prohibitively large. In the case of optimization levels O0 and O1, there are only 4 and 9 optimizations turned on by default allowing exhaustive enumeration. We measure the time to compile a method and *instrumented* each method with a fine-grain timer to record the amount of time spent executing that method. For optimization level O2, we could not exhaustive enumerate all possible optimization combinations (O2 has 20 optimizations), so for this level we choose to evaluate a set of 1000 randomly generated settings. The optimizations we control at each optimization level with our logistic regressors are described in Table 1.

The best settings were recorded in a vector **c** for each method. The size of **c** corresponds to the number of optimizations available at each level, i.e., size 4 for O0, 9 for O1, and 20 for O2.

3.3 Feature Extraction

Once we have examples of good optimization settings for different methods we would like correlate the settings with the characteristics of a method. To do this we need to describe each method in a sufficiently succinct form, to enable standard machine learning techniques to be employed.

Determining the properties of a method that predict an optimization improvement is a difficult task. As we are operating in a dy-

Optimizations	Meaning
Optimization Level O0	
BRANCH_OPTS_LOW	Turns on some simple branch optimizations
CONSTANT_PROP	Local constant propagation
CSE	Local common subexpression elimination
REORDER_CODE	Reorders basic blocks
Optimization Level O1	
COPY_PROP	Local copy propagation
TAIL_RECURSION	Tail recursion
STATIC_SPLITTING	Basic block static splitting
BRANCH_OPTS_MED	More aggressive branch optimizations
SIMPLE_OPTS_LOW	Type prop, Bounds check elim, dead-code elim, etc.
Optimization Level O2	
WHILES_INTO_UNTILS	Tries to turn whiles into untils
LOOP_UNROLL	Loop unrolling
BRANCH_OPTS_HIGH	Even more aggressive branch optimizations
SIMPLE_OPTS_HIGH	Additional pass of simple optimizations
LOAD_ELIM	Load elimination
REDUNDANT_BRANCH	Redundant branch elimination
SSA	Enter SSA form and perform SSA optimizations
EXPRESSION_FOLD	Expression folding
GLOBAL_COPY_PROP	Global copy propagation
GLOBAL_CSE	Global Common Subexpression elimination
COALESCE	Coalescing stage, requires SSA form

Table 1. This table describes all the optimizations investigated. These optimizations are all on by default for each of the different optimization levels. The optimizations at level O1 include all optimizations on at level O0 and optimizations at level O2 include all optimizations on at levels O1 and O0.

Feature	Meaning
bytecodes	Number of bytecodes in the method
locals space	Number of words allocated for locals
synch	Method is synchronized
exceptions	Method has exception handling code
leaf	Method is a leaf (contains no calls)
final	Method is declared final
private	Method is declared private
static	Method is declared static
Category	Fraction of bytecodes that ...
aload, astore	are Array Loads and Stores
primitive, long	are Primitive or Long computations (e.g., iadd, fadd)
compare	are Compares (e.g., lcmp, dcmpl)
branch	are Branches (forward/backward/cond/uncond)
jsr	are a JSR
switch	are a SWITCH
put/get	are a PUT or GET
invoke	are an INVOKE
new	are a NEW
arraylength	are an ArrayLength
athrow,checkcast,monitor	are an Athrow, checkcast, or monitor
multi_newarray	are a Multi Newarray
simple, long, real	are a Simple,Long, or Real Conversions

Table 2. Features of a Method. To reduce the length of the table several (different) features have been placed in logical groups.

namic compilation environment we chose features that are simple to calculate and which we thought were relevant. Computing these features requires a single pass over the bytecode of the method. We calculate features after inlining has been performed.

Table 2 shows the 26 features used to describe each method. The values of each feature will be an entry in the 26-element feature vector $\mathbf{x}$ associated with each method. The first 2 entries are integer values defining the size of the code and data of the method. The next 6 are simple boolean properties (represented using 0 or 1) of the method. The remaining features are simply the percentage of bytecodes belonging to a particular category. (e.g., 30% loads, 22% floating point, 5% yield points, etc.).

As an example, the feature vector for the method `compress.Compressor.cl_block()` is the following vector.

$$[108, 25, 0, 0, 0, 0, 1, 0, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0$$
$$0.12, 0.0, 0.08, 0.0, 0.0, 0.0, 0.2, 0.32, 0.08, 0.0]$$

In other words, there are 108 bytes in this method, 25 bytes allocated for locals, etc. We make no claim that this is the best set of features to describe a method. It is possible that an entirely different set of features would give better performance.

However, our logisitic regressors are able to learn automatically which features are most important to each of the different optimizations. And, since calculating all the features is very cheap (typically less than 1% of a method's compile time) there is no need to filter out unimportant features. The logistic regressor does this automatically.

Researchers have devised different features [1, 7, 6, 15, 20] that worked well on other optimization problems. For instance, Monsifort *et al.* [15] focus on loop structure as features in predicting whether to unroll or not. In future work, we will explore using other features such as those presented by Georges *et al.* [11] in combination with the features we used in this study.

3.4 Learning a classifier using logistic regression

We are now at the stage where we have, for each method, a feature vector $\mathbf{x}$ and a vector $\mathbf{c}$ which corresponds to the best settings for that particular method. Given a feature vector of a new method $\mathbf{x}$ we wish to develop a function, $\mathbf{f}$, that returns a good set of optimizations $\mathbf{c}$ to apply to it.

Classification

Our task can now be phrased as a classification problem: given a method's feature vector $\mathbf{x}$ should we apply a particular optimization or not? Given an $n = 26$ dimensional space of features, we wish to find a curve or hyperplane that separates those points where each optimization is turned on from those where it is off. To illustrate this, see the simple 2D example shown in Figure 3 where each point corresponds to a feature vector and we wish to find a line that separates them into classes.

Here the line $b + \mathbf{x}^T\mathbf{w} = 0$ forms the decision boundary, on the one side examples are classified as 1s, and on the other, 0s. The parameter b simply shifts the decision boundary by a constant amount. The orientation of the decision boundary is determined by $\mathbf{w}$ which represents the normal to the hyperplane.

However, in practice there does not exist a clean separation between points. Instead, we wish to associate with each point a *probability* of whether to apply an optimization. The simplest technique that achieves this is logistic regression [4] and is a probabilistic

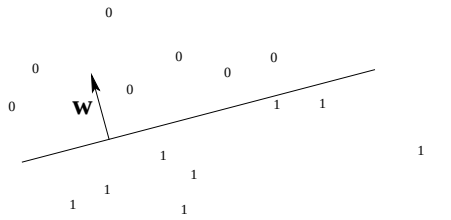


Figure 3. The linear separator decision boundary (solid line). For two dimensional data, the decision boundary is a line.

extension to linear regression. This provides us with a confidence measure as to how good our classification is.

Logistic Regression

We wish to determine a function **f** that gives the probability that a particular method should have a set of optimizations enabled. We have a set of up to to 20 optimizations to consider, **c**, but to ease presentation we initially consider the case of determining the probability that just one optimization **c** is enabled, i.e., $\mathbf{c} = 1$. Formally this is stated as:

$$p(c = 1|\mathbf{x}) = f(\mathbf{x};\mathbf{w}) \tag{1}$$

where $p(c = 1|\mathbf{x})$ is the probability that, given a feature vector **x**, optimization **c** is enabled or turned on and **f** is some function parameterised by the weights **w**. Since the function **f** represents a probability, it must be bounded between 0 and 1.

One of the standard choices of function is the sigmoid function, $\sigma(y) = 1/(1+\exp(-y))$. When the argument of the sigmoid function is positive, the probability that the input point **y** belongs to class 1 is above 0.5. The greater the argument value, **y**, is, the higher is the probability that it is classified as having the optimization setting *c* enabled. Similarly, large negative values of **x** will imply that *c* is disabled. Logistic regression based classification is described as:

$$p(c = 1|\mathbf{x}) = \sigma(b + \mathbf{x}^T\mathbf{w}) \tag{2}$$

where *b* is a constant scalar, and **w** is a vector of weights. Selecting the weights **w** allows the selection of the separating boundary orientation and a mechanism to state how confident we are in this boundary classification. The larger the weights, the more confident we are in the classification.

Training

Given that the sigmoid function is a good one to classify the data, we now have to derive or learn it. So, given the training data set *D*, gathered during the earlier exhaustive or random search, how can we adjust or learn the weights to obtain a good classification? Assuming that each of the *P* data points has been drawn independently the probability that the data belongs to a particular class c is given by a standard formula [4]:

$$p(D) = \prod_{j=1}^{P} p(c^j|\mathbf{x}^j) = \prod_{j=1}^{P} (p(c = 1|\mathbf{x}^j))^{c^j} \left(1 - p(c = 1|\mathbf{x}^j)\right)^{1-c^j} \tag{3}$$

where $\mathbf{x}^j$ is the *j*th ($j \in 1,\ldots,P$) feature vector selected from the training data and c^j is its corresponding best optimization setting for that feature vector. If $p(D) = 1$ then we have a perfect classification and the data is clearly separated. In practice due to the noisy data we will not achieve a perfect classification, but instead wish to get the best possible classification. If we adjust the weights to maximise $p(D)$ this will give the best decision hyperplane.

Each of these probabilities *p* is a function of the weight vector **w**, so we wish to choose **w** in order to maximise $p(D)$. As the values of *p* are small, it is common [4] to work with the $L = log p(D)$ to avoid rounding errors:

$$L = \sum_{j=1}^{P} c^j \log p(c = 1|\mathbf{x}^j) + (1 - c^j) \log \left(1 - p(c = 1|\mathbf{x}^j)\right) \tag{4}$$

and try to maximize it instead. (Maximizing $L = log(p(D))$ is equivalent to maximizing $p(D)$).

In the logistic regression model, we wish to therefore find the weights *w* to maximise:

$$L(\mathbf{w},b) = \sum_{j=1}^{P} c^j \log \sigma(b + \mathbf{w}^T\mathbf{x}^j) + (1 - c^j) \log \left(1 - \sigma(b + \mathbf{w}^T\mathbf{x}^j)\right) \tag{5}$$

Unfortunately, this can not be achieved analytically and is normally achieved by using an iterative solver based on gradient ascent, based on the partial derivatives of *L*.[2] The gradient is given by the following equation:

$$\nabla_{\mathbf{w}} L = \sum_{j=1}^{P} (c^j - \sigma(\mathbf{x}^j;\mathbf{w}))\mathbf{x}^j \tag{6}$$

The derivative with respect to the biases is as follows:

$$\frac{dL}{db} = \sum_{j=1}^{P} (c^j - \sigma(\mathbf{x}^j;\mathbf{w})) \tag{7}$$

In other words, select a value of **w** and update it in the direction of ascent. If $\nabla_{\mathbf{w}}$ is the partial derivative with respect to the vector **w** then we update our values of w and b as follows:

$$\mathbf{w}^{new} = \mathbf{w} + \eta \nabla_{\mathbf{w}} L \tag{8}$$

$$b^{new} = b + \eta \frac{dL}{db} \tag{9}$$

where η, the *learning rate* is a small scalar chosen small enough to ensure convergence of the method.

This is repeated until there is no further change and we have reached the maximum. At the end of this iterative process we have a set of weights and offset that gives the best classification on a given set of training inputs. It can then be used as a function which determines for a set of input features the probability of whether an optimization should be turned on or off.

3.5 Deployment

The final step involves installing the heuristic function in the compiler and using it during dynamic compilation. Each method that is compiled by the optimizing compiler is considered a possible candidate for all optimizations. We compute features for the method. If the heuristic function says we should optimize a method with a particular optimization, we do so. As an illustration, when applying the logistic regressor to the feature vector of `compress.Compressor.cl_block()`, it returns a value of

$$[1,0,1,1,0,0,0,1,1,1,1,1,1,1,1,0,1,1,1,0]$$

denoting which of the 20 optimizations to apply with the associated probabilities

[2] as $\sigma'(x) = \sigma(x)(1 - \sigma(x))$), partial derivatives are easy to calculate

Program	Description
compress	Java version of 129.compress from SPEC 95
jess	Java expert system shell
db	Builds and operates on an in-memory database
javac	Java source to bytecode compiler in JDK 1.0.2
mpegaudio	Decodes an MPEG-3 audio file
raytrace	A raytracer working on a scene with a dinosaur
jack	A Java parser generator with lexical analysis

Table 3. Characteristics of the SPECjvm98 benchmarks.

$$[0.7, 0.3, 0.8, 0.7, 0.4, 0.3, 0.1, 0.9, 0.7, 0.6,$$
$$0.6, 0.7, 0.9, 0.6, 0.6, 0.2, 0.6, 0.7, 0.7, 0.3].$$

This means that with a 70% probability BRANCH_OPTS_LOW should be applied and that with a 30% probability of CONSTANT_PROP should be applied. In other words apply branch optimizations, but do not apply local constant propagation for this method.

4. Infrastructure + Methodology

Here we describe the platform and benchmarks used as well as the methodology employed in our experiments.

4.1 Platform

We implement our learned heuristic in the Jikes Research Virtual Machine [3] version 2.3.3 for an Intel x86 architecture. The Intel processor is a 2.6GHz Pentium-4 based Red Hat Linux workstation with 500M RAM and a 512KB L1 cache. We used the FastAdaptiveGenMS configuration of Jikes RVM, indicating that the core virtual machine was compiled by the optimizing compiler, that an adaptive optimization system is included in the virtual machine, and the generational mark-sweep garbage collector was used.

4.2 Benchmarks

We examine two suites of benchmarks. The first is the SPECjvm98 benchmarks [19] which were run with the largest data set size (called 100). These benchmarks are described in Table 3.

The second set of programs consists of 5 programs from the DaCapo benchmark suite [5], ipsixql, and SPECjbb2000 and are described in Table 4. The DaCapo benchmark suite is a collection of programs that have been used for various different Java performance studies aggregated into one benchmark suite. We ran the DaCapo benchmarks under its default setting. We also included a program called ipsixql that performs XML queries and a modified version of SPECjbb2000 (hence, it is referred to as pseudojbb) that performs a fixed number of transactions. We refer to these 7 benchmarks collectively as DaCapo+.

4.3 Optimization Levels

We ran experiments under each of the three different optimization levels, that is, O0, O1, and O2 as well as the adaptive compilation scenario. When running under a particular optimization level we compile all methods using only the optimizations available at that level. The logistic regressor chooses which subset of these optimizations to apply to each method being compiled.

Although we specifically train only for the O0, O1, and O2 levels, we also evaluated the adaptive scenario which uses each of these three levels. Under the adaptive scenario, all dynamically loaded methods are first compiled by the non-optimizing baseline compiler that converts bytecodes straight to machine code without performing any optimizations. The resultant code is slow, but the compilation times are fast. The adaptive optimization system then uses online profiling to discover the subset of methods where a significant amount of the program's running time is being spent. These "hot" methods are then recompiled using the optimizing compiler. These methods are first compiled at optimization level O0, but if they continue to be important they are recompiled at level O1 and finally at level O2 if warranted. The individual optimization levels makeup the adaptive compiler and it is therefore important to tune these individual levels properly. When using logistic regression under the adaptive scenario, we used a logistic regressor trained for each of the three different optimization levels.

Program	Description
antlr	Parses one or more grammar files and generates a parser and lexical analyzer for each
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file
jython	Interprets a series of Python programs
pmd	Analyzes a set of Java classes for a range of source code problems
ps	Reads and interprets a PostScript file
ipsixql	Performs queries against a persistent XML document
pseudojbb	SPECjbb2000 modified to perform a fixed amount of work

Table 4. Characteristics of the DaCapo+ benchmarks.

4.4 Measurement

As well as the different compiler scenarios we also considered two different optimization goals namely: total time and running time. Total time is a combination of running and compilation time. As compilation is part of the total execution time for dynamic compilers then optimizing for total time will try to minimize their combined cost.

However, when the program is likely to run for a considerable length of time, it may be preferable for the user to reduce the running time at the expense of potentially greater compilation time. We therefore include running time for our benchmarks which is the execution time of the program without compilation time. Each benchmark was run multiple times and the minimum execution time is reported. [3]

4.5 Evaluation Methodology

As is standard practice, we learn over one suite of benchmarks, commonly referred to in the machine learning literature as the *training* suite. We then test the performance of our tuned heuristic over another "unseen" suite of benchmarks, that we have not tuned for, referred to as the *test* suite. These is achieved in two separate experiments.

Leave one out cross-validation We first use a standard approach to evaluate a machine learning technique called leave-one-out cross-validation on the SPECjvm98. Given the set of $n = 7$ benchmark programs, in training for benchmark i we train (develop a heuristic) using the training set from the $n - 1 = 6$ *other* benchmarks, and we apply the heuristic to the test set, the ith benchmark. So if we wish to test our technique on the `compress` benchmark, we first train using the results from all programs *except* `compress`. This way we never "cheat" in evaluating our heuristic on a program by having prior knowledge of that program.

[3] For total time, we ran each benchmark once and repeated this at least 5 times. For running time, we ran SPECjvm98 benchmarks 26 times removing the first run which includes the compilation costs of the program. For DaCapo+, which are longer running programs, we ran each program at least 5 times.

Testing on DaCapo+ To provide a different evaluation, we trained our heuristic on *all* the SPECjvm98 benchmarks and applied it to an entirely new benchmark suite, *DaCapo+*, of which the heuristic has no prior knowledge. The training set and the test suite *DaCapo+* are entirely distinct.

4.6 Training

For training, we exhaustively generated all optimization configurations for optimization levels O0 and O1. Given optimization level O0 consists to 4 optimizations and optimization level O1 consists of another 5 additional optimizations, this gives us 16 and 512 possible settings, respectively. Optimization levels O0 and O1 are subsets of O2, so we were able to use the exhaustively enumerated data for training a logistic regressor for O2. We randomly enumerated an additional 1000 optimization configurations to train the O2 logistic regressor. For each optimization configuration we recorded total execution times on a per method basis. These timings were used for training the logistic regressors where we select the optimization setting which gave the smallest total execution time.

Selection threshold Each logistic regression model returns the probability whether an optimization should be applied. We therefore have to make a decision as to the probability threshold beyond which we will apply the optimization. If $p = 0.5$ then the optimization can equally be applied or not - there is no conclusive decision. In our experiments, we make the conservative assumption that $p > 0.6$ before we apply an optimization. Some investigation showed the value 0.6 to be a reasonable value to use. Of course it is also possible to learn the ideal threshold, but this is beyond the scope of the paper.

5. Results

In this section, we apply our trained logistic regressors to our benchmark suites and compare their performance against using the default heuristic. The default heuristic is to apply all 4 optimizations at level O0, all 9 optimizations at level O1, and all 20 optimizations at level O2. For each of the three different optimization levels available in the Jikes RVM compiler(O0, O1, and O2), we have a specifically trained regressor that replaces the default heuristic.

We first discuss our results applying our logistic regressors to optimize SPECjvm98, then discuss our results optimizing DaCapo+. We note here that all timings include computation of features for each method which is a simple linear pass as well as the computation involved in applying our learned heuristic. The cost of computing features and applying our heuristic function is typically less than 1% of total compilation costs.

5.1 SPECjvm98

We now discuss the performance of our logistic regressors on the SPECjvm98 benchmarks relative to the default settings. The results are presented in Figures 4(a) through 4(d).

Opt Level O0: We first applied logistic regression to optimization level O0. Under this scenario, we reduce average total time down by 4%. Most notably we reduce the total time of mpegaudio by 23%.

It is worth noting that these results are similar to the results in Figure 1 where we show the results of applying the best setting found for each method. This provides evidence that our logistic regressor has learned the correct flag settings from this data set and is also an indication that the features used make our data linearly separable. Thus, because we are able to selectively apply optimizations to each method, we are inhibiting optimizations when they degrade performance and only applying them when they are beneficial.

We also get a significant reduction in running time of 5% on average. Again, we significantly reduce running time of mpegaudio by 26%. Along with mpegaudio, we are able to reduce running time of 3 other programs.

Opt Level O1: For optimization level O1, our learned models give us better total time and running time on average, 3% and 2% respectively.

Again, if we look at the results when applying the best flag found for each method, we see there is only a modest improvement we can achieve on total time or running time over the default.

Opt Level O2: For optimization level O2, we achieve significant improvements over the default. Our logistic regressor learned that a substantial number of optimizations at this level do not impact performance and therefore should not be applied.[4] Thus, it is possible to significantly reduce compilation time and total time over the default. This comes at no change in running time on average. Thus, we are able to reduce total time of this scenario substantially (by 29% on average) with no change in running time on average.

Adaptive: The standard model of execution used by today's modern Java JIT compiler is an adaptive scenario. Under the adaptive scenario, Jikes RVM uses *multi-level selective optimization*, that is, multiple optimization levels are used and the most important methods are optimized multiple times, each time with at successively higher optimization level. When a method is optimized at a particular level, the logistic regression trained for that level can be used in place of the default heuristic.

Under the adaptive scenario, our learned models reduces total time on average by 1%. This comes from improvements in total time for compress (3%), jess (3%), and javac (2%). We were also able to reduce the running time of most SPECjvm98 benchmarks, up to 5% for compress and javac. This leads to an average decrease of 2% over the default. There are several reasons why we are unable to improve performance under the adaptive scenario for these benchmarks. First, under the adaptive scenario, the optimizing compilers is only used for a small proportion of all methods compiled. Because most methods in this scenario are compiled with the baseline compiler we do not benefit from reductions in compile time. Second, the adaptive setting in Jikes RVM has been highly tuned for the performance of SPECjvm98 benchmarks. Therefore, there is little room for improvement with regards to running time on top of the extensive hand-tuning that has already been done.

5.2 DaCapo+

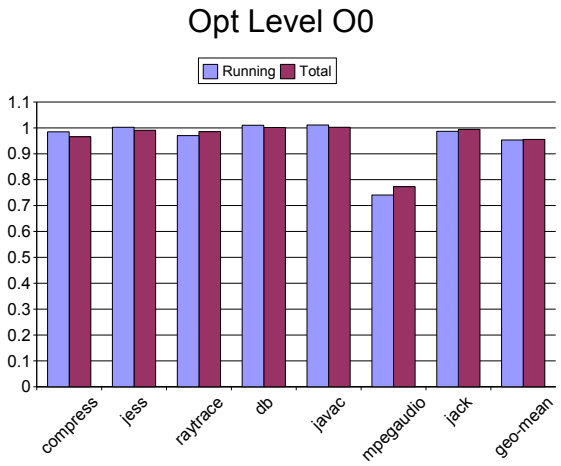
Next, we applied the logistic regressors that were trained using SPECjvm98 training data to the DaCapo benchmark suite and two additional benchmarks, ipsixql and pseudojbb. The results are presented in Figures 5(a) through 5(d).

For optimization level O0, we obtain average decreases of 3% for running time and 2% for total time. We get substantial decreases in running time for fop (9%), jython (5%), and antlr (8%) and for these benchmarks we also achieve some benefit in total time.

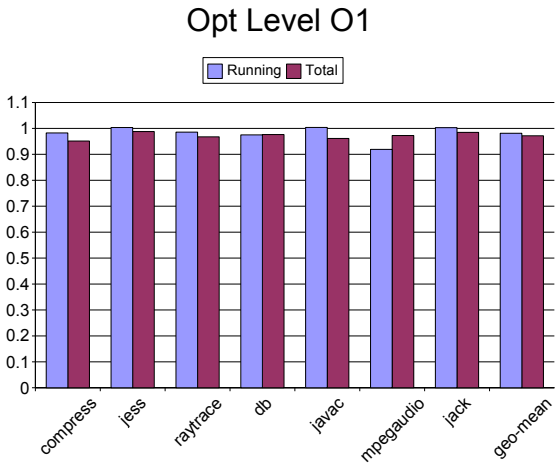
For optimization level O1, we get an even larger decrease in running time of 13% on average with a smaller decrease in total time of 2% on average. At this level, we can decrease the running time of jython significantly (11%) and we get a large decrease in running time for antlr at 63%. We do get a slow down for two programs, fop and pmd, however because of a large decrease from antlr our average is still significantly better than the default. Again, we see substantially improved performace with optimization level O2 using our regressors. We achieve a large decrease in average total time of 33% and an even more dramatic decrease in average running time of 56%.

Clearly many optimizations performed by default degrade performance. By selectively applying optimizations at level O2 (and

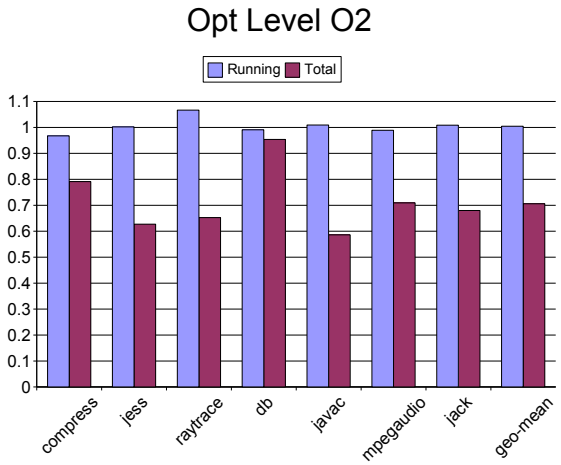
[4] We show this in more detail in Section 6



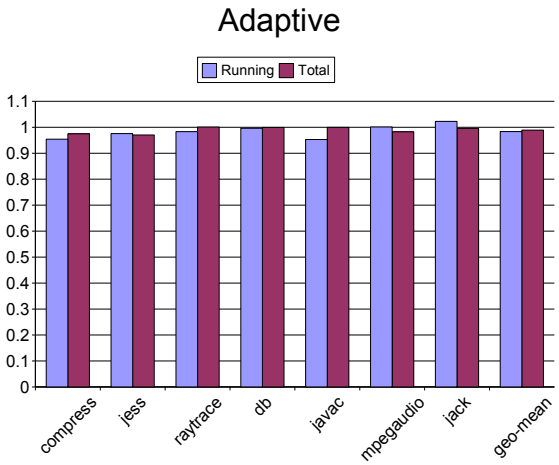
(a) Logistic Regressor for Opt Level O0 on SPECjvm98



(b) Logistic Regressor for Opt Level O1 on SPECjvm98



(c) Logistic Regressor for Opt Level O2 on SPECjvm98



(d) Logistic Regressor for adaptive on SPECjvm98

Figure 4. Performance of logistic regressors for the SPECjvm98 benchmarks. Table 1 lists the optimizations controlled by the regressor at each optimization level. The results are relative to the default setting, that is, a 1.0 indicates performance equal to the default setting and below 1.0 is performance better than the default.

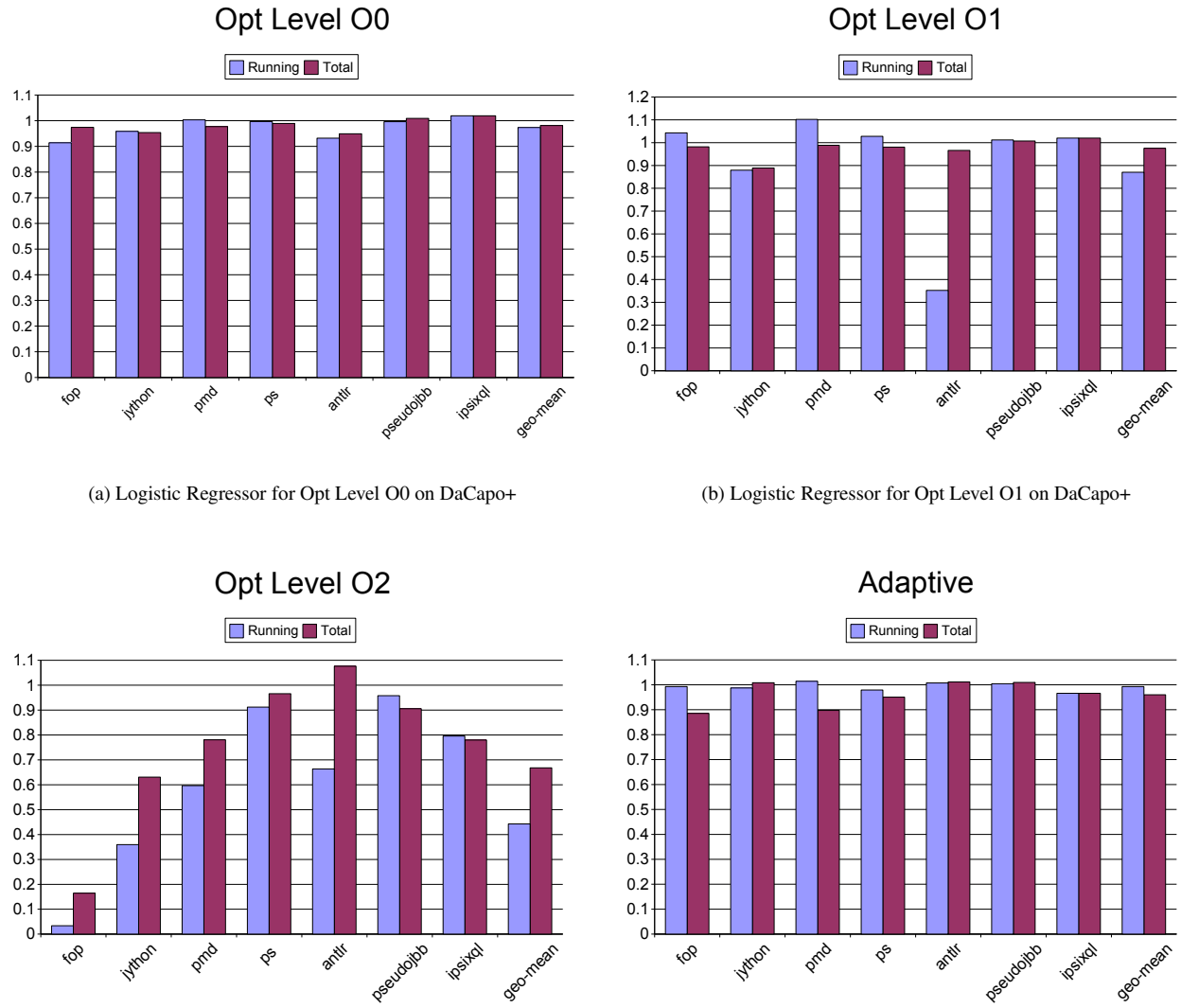
to a lesser extent at level O1), we can improve running and total time significantly. Note for optimization levels O1 and O2, we can decrease running time of DaCapo+ benchmarks much more than SPECjvm98 benchmarks. We believe this is due to the Jikes RVM compiler being highly tuned toward the SPECjvm98 benchmarks. In effect, the optimization heuristics in the Jikes RVM compiler have been specialized to the SPECjvm98 benchmarks. In contrast, using logistic regression allows us to construct heuristics that are more general and that can significantly improve performance on ”unseen” benchmarks.

Finally, we apply our learned models under the adaptive scenario to our DaCapo+ suite. For most benchmarks we get a decrease in total time. For fop and ps, we improve total time by at least 10% and for ps we get a 5% improvement. On average, we decrease total time by 4%. We get smaller decreases in running time leading to a 1% decrease on average.

6. Discussion

This section discusses the optimizations that are applied by the logistic regressors for the different optimization levels. Tables 5, 6, and 7 show the percentages that each optimization was applied at the different optimization levels for the *hot methods* of the SPECjvm98 benchmarks.[5] To calculate these percentages we counted the number of times each optimization was applied at a specific level when compiling the hot methods of a benchmark

[5] Here, we define the hot methods of a benchmark to be those methods that run long enough to trigger recompilation in adaptive mode.



(a) Logistic Regressor for Opt Level O0 on DaCapo+

(b) Logistic Regressor for Opt Level O1 on DaCapo+

(c) Logistic Regressor for Opt Level O2 on DaCapo+

(d) Logistic Regressor for adaptive on DaCapo+

Figure 5. Performance of logistic regressors for the DaCapo+ benchmarks. Table 1 lists the optimizations controlled by the regressor at each optimization level. The results are relative to the default setting, that is, a 1.0 indicates performance equal to the default setting and below 1.0 is performance better than the default.

and divided by the total number of hot methods compiled for that benchmark.

The tables show that the importance of each optimization differs for the different benchmarks. For instance, Table 5 shows that CONSTANT_PROP is applied to 81% of raytrace's hot methods and is therefore important to that benchmark while it is only applied to 57% of jack's hot methods and is therefore not as important for that benchmark. On the other hand, CSE is more important to jack (applied 84%) than raytrace (applied 66%). This indicates that certain optimizations are more important for some benchmarks and not for others further motivating the need for method-specific optimization configurations.

Tables 5 and 6 show that the importance of an optimization depends on the other optimizations it might enable or disable at that optimization level. For example, BRANCH_OPTS_LOW is not very important at optimization level O0 (with the exception of javac). However, at optimization level O1 this optimization becomes very important to most of the benchmarks. This is due to the enabling effect BRANCH_OPTS_LOW has on other optimizations at this level.

Tables 6 and 7 show that many optimizations are not important for all benchmarks. These optimizations can perhaps be removed or turned off by default with little or no effect on overall performance. The tables also show which optimizations are consistently important for all benchmarks. This can give an indication to JVM designers which optimizations should be given highest priority when developing new JIT compilers.

Optimizations	compress	jess	raytrace	db	javac	mpegaudio	jack
BRANCH_OPTS_LOW	20	0	11	12	60	22	24
CONSTANT_PROP	40	25	81	75	95	70	57
CSE	20	75	66	88	81	78	84
REORDER_CODE	0	16	59	75	81	44	65

Table 5. The percent of level O0 optimizations applied to the hot methods of each benchmark.

Optimizations	compress	jess	raytrace	db	javac	mpegaudio	jack
BRANCH_OPTS_LOW	75	4	77	100	95	73	54
CONSTANT_PROP	50	80	85	0	69	29	63
CSE	100	91	97	100	98	100	100
REORDER_CODE	50	4	82	50	50	67	97
COPY_PROP	100	100	100	100	100	100	100
TAIL_RECURSION	0	24	87	100	89	89	34
STATIC_SPLITTING	0	27	31	0	19	6	20
BRANCH_OPTS_MED	100	11	90	100	98	97	69
SIMPLE_OPTS_LOW	100	18	36	100	89	95	100

Table 6. The percent of level O1 optimizations applied to the hot methods of each benchmark.

Optimizations	compress	jess	raytrace	db	javac	mpegaudio	jack
BRANCH_OPTS_LOW	100	78	88	100	96	98	91
CONSTANT_PROP	0	11	25	50	24	49	60
CSE	50	2	35	17	26	70	9
REORDER_CODE	100	27	70	83	7	92	97
COPY_PROP	100	100	100	100	100	100	100
TAIL_RECURSION	25	96	80	83	96	54	69
STATIC_SPLITTING	0	7	35	0	98	38	3
BRANCH_OPTS_MED	50	27	32	33	0	49	26
SIMPLE_OPTS_LOW	0	22	57	50	95	10	17
WHILES_INTO_UNTILS	100	93	40	33	4	86	69
LOOP_UNROLL	50	0	5	0	0	0	14
BRANCH_OPTS_HIGH	50	98	92	100	91	57	100
SIMPLE_OPTS_HIGH	0	0	2	0	1	0	23
LOAD_ELIM	100	100	100	100	16	71	100
REDUNDANT_BRANCH	100	89	100	100	97	100	100
SSA	100	100	100	100	100	100	100
EXPRESSION_FOLD	0	0	0	0	25	25	0
GLOBAL_COPY_PROP	25	2	20	0	3	6	11
GLOBAL_CSE	0	4	12	17	33	41	23
COALESCE	50	0	8	0	2	30	6

Table 7. The percent of level O2 optimizations applied to the hot methods of each benchmark.

7. Related Work

There have been a number of paper aimed at using machine learning to tune individual optimization heuristics.

Moss *et al.* [16] published one of the first papers showing that machine learning could successfully construct effective and efficient compiler heuristics. They used supervised learning techniques to construct a heuristic function for instruction scheduling. The automatically constructed heuristic was able find schedules that performed as well as a highly-tuned hand-crafted instruction scheduler.

Calder *et al.* [6] used supervised learning techniques, namely decision trees and neural networks, to induce static branch prediction heuristics. Our learning methodologies are similar, but there are important differences. First, they began with a rich set of hand-crafted heuristics from which to derive their features. In contrast, we had no pre-existing heuristics from which to draw features. Second, their technique made it inherently easy to determine a label for their training instances. The optimal choice for predicting a branch was easily obtained by instrumenting their benchmarks to observe each branch's most likely direction. We obtained our labels using method timings as we discuss in Section 4.6. Also, because our timing measurements are imprecise and we do not take interaction effects of different methods into account, it is impossible to determine the optimal choice of whether or not to apply an optimization.

Stephenson *et al.* [20] used genetic programming (GP) to tune heuristic priority functions for three compiler optimizations: hyperblock selection, register allocation, and data prefetching within the Trimaran's IMPACT compiler. For two optimizations, hyperblock selection and data prefetching, they achieved significant improvements. However, these two pre-existing heuristics were not well implemented and most of the improvements came from producing 400 random heuristics and choosing the best heuristic from this group. The authors even note that turning off data prefetching completely is preferable and reduces many of their significant gains. For the third optimization, register allocation, iterating the GP improved over the initial population. However, for this optimization they were only able to achieve on average a 2% increase over the manually tuned heuristic. Stephenson et al. [21] use machine-learning to characterize the best unroll loop factor for a given loop nest, and improve overall by 1% over the ORC compiler heuristic with SWP enabled. Both of these approaches are successful in automatically generating compiler heuristics for a single optimization.

Cavazos *et al.* [7] describe an idea of using supervised learning to control whether or not to apply instruction scheduling. They induced heuristics that used features of a basic block to predict whether scheduling would benefit that block or not. Using the induced heuristic, they were able to reduce scheduling effort by as much as 75% while still retaining about 92% effectiveness of scheduling all blocks. However, they were unable to reduce the total execution time for the SPECjvm98 benchmark suite.

Monsifrot *et al.* [15] use a classifier based on decision tree learning to determine which loops to unroll. They looked at the performance of compiling Fortran programs from the SPEC 95 benchmark suite and some computational kernels using g77 for two different architectures, an UltraSPARC and an IA64. They showed an average improvement over g77's hand-tuned unroll heuristic of 3.1% and 2.7% on the IA64 and UltraSPARC, respectively.

Lagoudakis *et al.* [13] describe an idea of using features to choose between algorithms for two different problems, order statistics selection and sorting. The order statistics selection problem consists of an array of n (unordered) numbers and some integer index i, $1 <= i <= n$. The problem involves selecting the number that would rank i-th in the array if the numbers were sorted in ascending order. The authors used reinforcement learning to choose between two well-known algorithms: Deterministic Select and Heap Select. The learned algorithm outperformed both these algorithms at the task of order statistics selection. The second problem they look at is the *sorting problem*, that is, the problem of rearranging an array of n (unordered) numbers in ascending order. Again, the authors used reinforcement learning to choose between two algorithms: Quicksort and Insertion Sort. The learned algorithm again was able to outperform both of these well-known algorithms.

Rather than optimizing a single heuristic, others have looked at searching [22, 10, 9, 8, 12, 17, 14] for the best set or sequence of optimizations for a particular program. Cooper *et al.* [9] propose a number of algorithms to solve the compilation phase ordering problem. Their technique searches for the best phase order of a particular program. Such an approach gives impressive performance improvements but has to be performed each time a new application is compiled. While this is acceptable in embedded environments, it is not suitable for dynamic compilation.

Kulkarni *et al.* [12] introduce techniques to allow exhaustive enumeration of all distinct function instances that would be produced from the different phase-orderings of 15 optimizations. This exhaustive enumeration allowed them to construct probabilities of enabling/disabling interactions between the different optimization passes. Using these probabilities, they constructed a *probabilistic batch compiler* that dynamically determined which optimization should be applied next depending on which one had the highest probability of being enabled. This method however does not con-

sider the benefits each optimization can potentially provide when applied. In contrast, we train our logistic regression on the best optimizations found for each method, and therefore our technique learns which optimizations are beneficial to apply to "unseen" methods with *similar* characteristics. However, the techniques presented in this work would allow a larger exploration of the optimization space than we attempted. By exploring a larger part of the search space, we would likely improve the data used for training our logistic regressors.

Pan *et al.* [17] partitioned a program into *tuning sections* and then developed fast techniques to find the best combination of optimizations for each of these tuning section. They are able to reduce the time to find good optimization settings from hours to minutes. This technique, although useful in static compilers (especially those targetting embedded processors), is not applicable to dynamic compilers where large optimization times can easily outweight any benefits gained from the optimizations. However, these techniques could also be beneficial during the training data generation stage of our logistic regressor technique. Specifically, the technique to test different optimization settings on a tuning section during a single run of the program would allow us to increase the number of optimization settings we evaluate. This would also improve the quality of the training data we used for our logistic regressors.

Agakov *et al.* [1] show that the iterative compilation search space can be reduced by learning from other programs. They construct a set of probabilities, called search distributions, for a set of training programs and use features to choose with nearest neighbor which search distribution to use for a new program. However, this approach is to select optimization configurations for a whole program and still requires multiple runs of the program to achieve significant improvement.

In the area of predictive modelling, Zhao *et al.* use manually constructed cost/benefit models to predict whether to apply PRE or LICM[24]. They achieve 1% to 2% improvement over always applying an optimization, but at a cost of greatly increasing compilation time (by up to 68%). Because it is expensive to apply, their predictive models would not be beneficial in a dynamic compilation setting. Also, their models appear to be quite complicated and have to be manually constructed. Our models, on the other hand, are simple and automatically constructed using machine learning.

Yotov *et al.* [23] describe a model-based approach for optimizing BLAS libraries. They show that using a model-based approach to evaluate the performance of an optimization can be as effective as empirical evaluation. Again, their models are complicated and require manual tuning. In contrast, our regressor models are automatically constructed and have the potential to outperform hand-tuned models.

Lau *et al.* [14] present an online framework, called *performance auditing*, that allows the evaluation of the effectiveness of optimization decisions. The framework allows for online empirical optimization, which improves the ability of a dynamic compiler to increase the performance of optimizations while preventing performance degradations. Instead of using models to predict an optimization's performance they compile different versions of the same method with different optimization settings and then run each of these different versions evaluating their performance empirically on the real machine. Combining these techniques with the techniques presented in this paper would be interesting future work. For example, one could use machine learning to create a small set of optimization settings, which could be explored online using performance auditing.

Georges *et al.* [11] present a technique for measuring processor-level information gathered through performance counters and linking that information to specific methods in a Java program. They study method-level phase behavior of Java applications to identify methods that exhibit similar or dissimilar behavior within the phases. They characterize methods using a number of performance counter events such as cache miss rates, TLB miss rates, branch misprediction rates, etc. This information can allow developers to identify performance bottlenecks and to gather insights on how a Java application interacts with the VM. For our logistic regression techniques, we trained using only static features of a method which are cheaper to collect than dynamic performance counter information. However, in future work, we would like to investigate whether dynamic features (perhaps in combination with static features) could improve the predictions of our machine learning techniques.

8. Conclusions

This paper has shown that method-specific optimization settings can give significant performance improvements within the Jikes RVM JIT compiler. It has also demonstrated that a simple machine learning technique can automatically derive a predictive model that gives significant performance improvements over existing schemes. We show total execution time reductions of 25% and 51% on the SPECjvm98 and DaCapo+ benchmark suites respectively. To our knowledge this is the first paper to demonstrate that a predictive model trained with machine learning can be successfully used as a method-specific optimization strategy within a dynamic compiler. Future work will investigate learning the best ordering of transformations on a per method basis and applying this approach to the adaptive compilation setting. We would also like to experiment with different kinds of features, such as dynamic performance counter features, and perhaps investigate the use of online empirical evaluation in order to evaluate a few optimization settings for each method that a learned model predicts as being promising.

9. Acknowledgements

Thanks to Dries Buytaert for providing a patch allowing us to instrument methods to collect fine grain timing information. Also, thanks to Felix Agakov and Marc Toussaint for suggesting we use logistic regression for this problem. Finally, thanks to the anonymous reviewers for their helpful comments on drafts of this paper.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussiant, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Fourth Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 295–305, New York City, NY, March 2006.

[2] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, 2004.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb. 2000.

[4] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 2005.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking

development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[6] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zoren. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, January 1997.

[7] J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 183–194, Washington, D.C., June 2004. ACM Press.

[8] K. D. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, , and T. Waterman. Acme: adaptive compilation made efficient. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[9] K. D. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, , and T. Waterman. Searching for compilation sequences. Tech. report, Rice University, 2005.

[10] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[11] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in java workloads. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 270–287, New York, NY, USA, 2004. ACM Press.

[12] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Exhaustive optimization phase order space exploration. In *Fourth Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 306–318, New York City, NY, March 2006.

[13] M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 511–518, Stanford, CA, June 2000. Morgan Kaufmann.

[14] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: using hot optimizations without getting burned. In *Proceedings of the ACM SIGPLAN '06 Conference on Programming Language Design and Implementation*, pages 239–251, Washington, D.C., June 2006. ACM Press.

[15] A. Monsifrot and F. Bodin. A machine learning approach to automatic production of compiler heuristics. In *Tenth International Conference on Artifi cial Intelligence: Methodology, Systems, Applications (AIMSA)*, pages 41–50, Varna, Bulgaria, September 2002. Springer Verlag.

[16] J. E. B. Moss, P. E. Utgoff, J. Cavazos, D. Precup, D. Stefanović, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *Proceedings of Neural Information Processing Systems 1997 (NIPS*97)*, Denver CO, Dec. 1997.

[17] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *IEEE PACT*, Seattle, WA, September 2006. IEEE Computer Society.

[18] R. Pinkers, P. Knijnenburg, M. Haneda, and H. Wijshoff. Statistical selection of compiler options. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 494–501, 2004.

[19] Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. *SPEC JVM98 Benchmarks*, 1998.

[20] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 77–90, San Diego, Ca, June 2003. ACM Press.

[21] M. Stephenson and S. P. Amarasinghe. Predicting unroll factors using supervised classification. In *Third Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 123–134, 2005.

[22] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *First Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 204–215, 2003.

[23] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 63–76, San Diego, Ca, June 2003. ACM Press.

[24] M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *Third Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 317–327, 2005.