This is a preprint of an article that will appear in Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC), 8 ms, Rio de Janeiro, Brazil. October 2010. IEEE Press

Using Support Vector Machines to Learn How to Compile a Method

Ricardo Nabinger Sanchez[†] José Nelson Amaral[†] Duane Szafron[†] Marius Pirvu[‡]
Mark Stoodley[‡]

Abstract

The question addressed in this paper is what subset of code transformations should be attempted for a given method in a Just-in-Time compilation environment. The solution proposed is to use a Support Vector Machine (SVM) to learn a model based on method features and on the measured compilation and execution times of the methods. An extensive exploration phase collects a set of example compilations to be used by the SVM to train the model. This paper reports on a work in progress. So far, linear-SVM models, applied to benchmarks from the SPECjvm98 suite, have not outperformed the compilation plans engineered by the development team over many years. However the models almost match that performance for the `javac` *benchmark.*

1 Introduction

Modern Just-in-Time (JiT) compilers have many code transformations that could be applied to each method that is compiled. However, with compilation competing with the execution of the application for resources, code transformations for each method must be selected carefully. Methods are compiled at different levels of optimization based on an estimation of the future execution frequency of the method. A significant amount of effort is spent in the design of a commercial compiler to determine the combination of code transformations that should be included in each one of these levels of compilation. However, once a method is selected for compilation at a certain level, all the code transformations that are applicable at that level are attempted.

† Department of Computing Science, University of Alberta, Edmonton, AB, Canada
‡ IBM Toronto Software Laboratory, Markham, ON, Canada
This research is supported by fellowships and grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Collaborative Research and Development (CRD) program, and by the IBM Center for Advanced Studies. We thank Kevin Stoodley for the initial discussions and encouragement for this project and many developers at the IBM Toronto Software Laboratory for insights and assistance along the way.

The premise of this paper is that one-size-fits-all may not be the best approach when deciding which code transformations to apply to a given method. It is possible that some code transformations have no effect in some methods, and others may even result in slower code. We propose that it is sometimes possible to predict which code transformations should be attempted in a given method based on a set of features that can be easily extracted from the method before dynamic compilation takes place.

This paper reports on continuing research that investigates the potential compilation and execution time reductions that may be achieved when an SVM is used to predict which transformations should be used. This experimental evaluation uses a development version of Testarossa, the IBM™ JiT commercial compiler for Java. Section 7 compares this approach with the closely related work by Cavazos and O'Boyle [2] and by Eeckout *et al.* [7].

The implementation of the infrastructure for learning in a large-scale commercial compiler is non-trivial. For instance, code transformations are not isolated stand-alone functions that can be composed at will. In Testarossa, collections of individual transformations are organized in a transformation pipeline. Two changes required for the learning framework interfere with these transformation pipelines: (a) selecting a combination of transformations that were not intended to be used by the compiler engineers; and (b) instrumenting execution to investigate the space of code transformations. The transformations in a pipeline are tailored to a set of compilation plans that is deployed with the commercial compiler. Therefore the composition of transformations is constrained by engineering decisions in the design of the compiler.

Even though there are readily available profiling solutions, they tend to be excessively intrusive (sometimes profiling the compiler itself) and tend to impose an overwhelming overhead that dramatically changes the behavior of Testarossa. Keeping this overhead to a low level required a customized infrastructure for both the instrumentation mechanism and the storage of collected data. Our instrumentation approach incurs 124 CPU cycles (on average) per measurement instance. Our profiling infrastructure generates no ex-

tra I/O activity while the compiler is operating because it saves collected data to disk only after the application completes execution.

After the model is learned, there is no instrumentation or additional memory costs for compilations, only the time required to communicate with the model and for it to perform the classification, averaging $214\ \mu s$ per request in our setup.

This paper reports on a work in progress. The principal contributions so far include:

- We are the first group to develop a framework to enable the application of SVM to learn method-specific compilation strategies in a commercial compiler environment.

- We describe the complete framework to integrate learning in the compiler. This framework can be used with alternative learning strategies to create different prediction models.

- Our preliminary experimental results indicate that machine-learned method-specific compilation plans *can* match the hand-tuned adaptive compilation strategies in a commercial compiler.

Section 2 gives a brief overview of the organization of the compiler. Section 3 contains a brief review of SVMs. The data collection process is presented in Section 4. Section 5 describes the training of a model. Section 6 describes the experimental evaluation of the technique and Section 7 discusses related work.

2 Organization of the JiT Compiler

Testarossa is a state-of-the-art JiT compiler employed in the IBM J9™ Java Virtual Machine (JVM) [5, 8]. Its goal is to improve the performance of Java applications by converting bytecodes into native code during program execution. A balance needs to be struck between the overhead and the benefits resulting from JiT compilation because this conversion is done at runtime. Therefore, Testarossa focuses its efforts only on methods that are frequently executed.

In Testarossa, a method can be compiled multiple times at different optimization levels. Higher optimization levels are expected to generate faster executing code at the expense of more compilation time. Adjectives normally associated with temperature (cold, warm, hot, very hot, scorching) are used to refer to estimates of the frequency of execution of a method and to the compilation levels. These estimates, based on a combination of sampling and invocation counting mechanisms, are used to decide how much compilation effort to invest in attempting to increase the execution speed of a particular method.

Testarossa is composed of four major building blocks: (1) the *IL Generator* converts bytecodes into an intermediate language (IL) representation that is more amenable to optimizations; (2) the *Optimizer* applies code transformations to improve the quality of the IL code;[1] (3) the *Code Generator* transforms the optimized IL into native code (there is one code generator for each target platform x86, PPC, s390, MIPS, ARM, etc.); (4) the *Compilation Control* block makes compilation decisions (what methods to compile, when to compile them and at what level). Of these four building blocks, the Optimizer consumes the most time and requires the most memory.

Each compilation level has an associated compilation-strategy table that defines both the code transformations to be applied to a method and the order in which to apply them. In principle, the same set of code transformations is applied to all methods compiled at a given level. However, to conserve compilation resources, some transformations are associated with a conditional flag that controls their application (e.g. loop transformations are not attempted on a method that is known to have no loops). The compilation strategies are quite rich in transformations. For example, the cold strategy includes over 20 transformations, whereas the scorching one has more than 170. This number includes repeated applications of the same transformation. For instance, *dead tree elimination* is performed multiple times to clean-up after other code transformations are applied.

JiT compilations are performed in the background by a dedicated thread. Most compilations are asynchronous to the application's execution. A small subset may pause an application thread to wait for the compilation of a method. A case in point is when a compiled body that employs the *invariant-argument-pre-existence* transformation [3] becomes invalid due to an incompatible change in the JVM's class hierarchy.

3 A Primer on Support Vector Machines

This section presents a brief primer of SVMs, which are thoroughly described in the literature, to help the reader understand the remainder of the paper. It also discusses limitations of available implementations of SVMs that hinder their use for problems with large training sets.

SVMs are statistical learning models that work by finding maximum separation margins within a data set [6]. Each data point is composed of a feature vector that represents observations (*e.g.*: code size and presence of loops in a method to be compiled) and a label denoting its class. In our case, each class represents a different set of code transformations that can be applied to a method. The training of an SVM with a data set produces a matrix of weights of

[1] These transformations are very complex and are also referred to as "optimizations" in the literature.

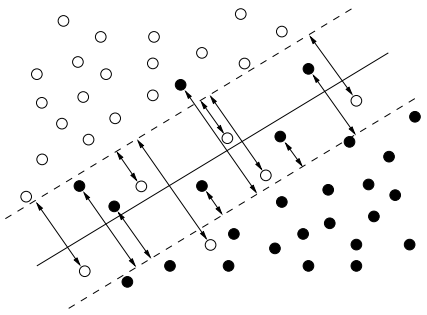

Figure 1. An example of a non-separable case, where some amount of miss-classification (arrows) based on the margins (dashed lines) must be allowed during hyperplane placement (solid line).

the contributions from each element of the feature vector to the classification function. Classifying an unseen feature vector consists of computing the product of the matrix with the feature vector to produce a vector with one component for each class. The component representing each class can be used to compute the relative confidence that the feature vector should be classified as belonging to that class. The simplest SVMs deal will a single class, and classification consists of deciding whether a data point belongs to that single class or not. For a single-class SVM, the classification result is given by the sign of the feature vector applied to the classification function.

The SVM implementation used in this study, LIBLINEAR [4], uses a one-versus-all approach to make class predictions. The one-versus-all approach trains one classifier for each class, which computes the confidence that the data point belongs to a given class, and the class with the highest confidence is selected. The confidence in the one-versus-all approach is the distance from the data point to the given classifier boundary. The boundaries of a classifier are hyperplanes separating the classes in the feature space and are found during the training of the model.

It is often the case that the data is not linearly separable in the original feature space. For those cases, SVMs employ *kernel functions* that map the original features into a higher-dimensional space, where they can be separated using hyperplanes. There are many kernel functions in the literature, and the most common ones are the Radial Basis Function (RBF) and d-th Degree Polynomial [6]. An iterative optimization process searches for the positioning of the hyperplanes that maximizes the margins of separation of the data, as illustrated in Figure 1. The figure shows an example where some amount of miss-classification is inevitable. The amount of miss-classification (arrows) in an SVM model is given by a cost parameter C specified by the user. C specifies how large the margins (dashed lines) can be from the separating hyperplane (solid line). Moreover, C also influences the orientation of the hyperplane because this orientation depends on how the overlapping data points[2] will shift the margins.

For larger data sets, either with a large number of samples and/or features, using a non-linear kernel may be impractical. First, non-linear kernels often require more memory and longer training times. Second, the resulting models are very large (several gigabytes in our experience). Large models are too slow to be used online during just-in-time compilation. Third, the point of using a non-linear kernel is to increase the separability of the data by projecting it into a higher dimension. However if the data is already in a high-dimensional space, moving it to a higher dimension may not have much effect in its separability. As a consequence, for high-dimensional data a non-linear kernel may exhibit similar performance to a linear kernel. For problems involving many data samples and a large number of features LIBLINEAR employs an identity kernel that acts as a linear one [9].

The number of features in the most common applications of SVM is in the 20–30 range. In this study, 81 features are used to describe each method. The set of features collected for each method include counters (*e.g.*: number of parameters received), attributes (*e.g.*: presence of loops), and distributions computed in a single pass over the intermediate representation of the method. These distributions characterize the operations and types of operands for the method.

4 Data Collection

For data collection, a benchmark is run with the JiT system executing in experimentation mode. In this mode, a method is selected for recompilation when it reaches a certain invocation threshold since the last compilation.[3] The value of this threshold is estimated for each method based on its first 8 invocations after it is initially compiled. The goal is to balance this threshold for both long and short running methods, ranging from 50 to $50,000$ invocations. In most cases, this threshold range allows methods to accumulate approximately 10 *ms* of execution between compilations.

Once the method is selected for recompilation, Testarossa selects a compilation plan based on the optimization level. In experimentation mode, code transformations are randomly removed from the plan, according to a *modifier*, to explore the space of possible code transformation

[2] Points of a given class located in a region populated by points belonging to other classes.

[3] If Testarossa decides to promote the optimization level for the method, the exploration will continue on the next level in the same way.

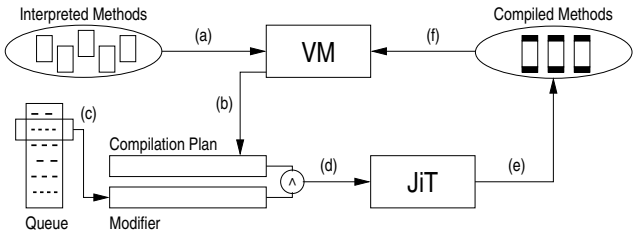

Figure 2. Data Collection during training.

combinations. Whenever the compiled method is executed, the running time of the method is recorded along with an identifier for the compilation plan used. The average running time from all invocations of the method compiled with a given plan is then combined with a weighted value for the compilation time to create a training point. This data is stored in compact structures that are kept in memory.

An alternative for the exploration of the space of code transformations would be to generate a random set of transformations to be applied when a method is compiled. However, because this is a complex compilation system developed over many years in an industrial setting, there are many dependencies between code transformations that would be violated by such an approach. Thus, the more conservative alteration of existing compilation plans through the random removal of code transformations is a safer alternative that implicitly satisfies dependencies between code transformations. The disadvantage of this approach is that it limits the exploration of the space of code transformations and thus might fail to discover combinations that were not considered by the compiler designers.

The block diagram in Figure 2 illustrates the data collection process. Before the execution of the program starts, a list of modifiers for compilation plans is created. Each modifier will be used for the compilation of k distinct methods.[4] Whenever a given method is recompiled, a different modifier is used. The number of modifiers to pre-compute for a given benchmark is determined experimentally by running the application once to measure the number of plans effectively used.[5] Pre-computing the modifiers has the advantages that it is trivial to (i) track which plans were used on a given method to avoid repetition and to (ii) apply the same modified plan to the compilation of different methods.

The first compilation of a method (Figure 2(a)) is triggered by the policies implemented in the virtual machine. Once a method is selected for compilation (b), the JiT data collection process starts. A modifier from the pre-computed list (c) is used to modify the compilation plan for the optimization level selected by the VM (d). A modifier does not change the order of the transformations. Moreover, the elimination of transformations from a plan must obey dependency constraints between the code transformations. For instance, some clean-up transformations are never disabled because the absence of a clean-up might lead to the generation of invalid code. Once the compilation is completed, the instrumented method is placed in the pool of compiled methods (e). This process is also used for methods selected for recompilation (f), either by a VM decision (*e.g.*: the method is invoked frequently enough to be recompiled at a higher optimization level) or due to a recompilation requested after its invocation count reaches the recompilation threshold used by the instrumentation mechanism.

5 Learning a Model

The goal is to create a model that maps a set of method features into a successful compilation plan. Therefore this model should be trained with a collection of such plans. The random exploration of the space of possible compilation plans yields both good and bad compilation plans. Thus the first step in the creation of the model is to rank the explored compilation plans to select the most effective ones that will be used to train the model.

The data preparation for the training of the model uses the following ranking function:

$$V_i = \frac{R_i}{I_i} + \frac{C_i}{T_h} \quad (1)$$

where R_i is the total running time for all invocations of method i between two compilations of the method; I_i is the number of invocations since the last recompilation; C_i is the compilation time for method i; and T_h is the triggering value used by Testarossa for recompiling at compilation level h (h is used to reflect the "hotness" level). The value V_i is used to rank all compilation plans generated for method i at compilation level h. The top t compilation plans for each method at each compilation level are selected to train the model. The set of plans is further tuned by enforcing a cut-off value relative to the best plan in the set.[6]

After the ranking is complete, the data points that are selected as input for the training are normalized to ensure that all data values are in the $[0, 1]$ range. Normalization is not strictly required for the training of SVMs, but it is recommended to improve numerical stability [9] and to improve the chances that the iterative training process will converge to a solution in a reasonable amount of time. At runtime, when the model is used to select compilation plans, the compiler will be producing raw data values. Therefore the

[4]The experiments reported in this paper use $k = 50$, which was determined empirically.

[5]For the analysis described in this paper using SPECjvm98 benchmarks [13], 8192 dynamic compilation plans were necessary for all applications in the suite using the largest inputs and 10 benchmark iterations.

[6]In the experiments reported, $t = 3$ and the relative cut-off is 95%.

normalization parameters are saved for use by the compiler during production runs.

The specification of a compilation plan is encoded in 58 bits for the version of the Testarossa compiler used in this research. Using this binary representation to represent each plan in the model created by the SVM would result in 2^{58} possible plans. However not every 58-bit binary combination is a legal plan. Moreover, LIBLINEAR assumes that a class is encoded in 32 bits. The solution is to remap the plans into a new space. Remapping is accomplished by assigning a sequential integer number, starting from 1 (another requirement of LIBLINEAR) to each unseen plan and to maintain an index file to map from these unique identifiers to the compilation plans. This simple approach is transparent to the learning process and has negligible overhead.

An important parameter to specify for an SVM is the misclassification cost C. This parameter determines how the SVM should deal with misclassifications while trying to find the best separating margins between classes. The value of C is especially important for larger data sets where multiple class overlappings are likely to occur. In such situations some amount of misclassification is necessary to enable the model to converge to a solution.[7]

6 Experimental Evaluation

The overall strategy is to train a separate model for each compilation level (cold, warm, hot, and scorching). This paper presents results only for the hot optimization level. Therefore the experimental evaluation using the various models are compared both with the original compiler and with a version in which all methods are compiled at the hot level — which we call the *hot compiler*. In Figures 3 and 4 the hot compiler is the baseline. This experimental evaluation resulted in the following main findings:

- The overhead of applying the learned model during JiT compilation is very low for an SVM using a linear kernel, accounting for less than 0.1% of the total running time of the application.
- Training times for a widely used set of benchmarks are manageable in a few minutes after data set ranking.
- The ability of Testarossa to adapt the level of optimization produces significant speedups in relation to the hot compiler. For throughput performance, only in one benchmark, *javac*, did the learning model approximates the performance of the adaptive compiler.
- When considering start-up performance, the learning model successfully approximates the performance of the adaptive compiler. For *javac*, the learning model is able to outperform the adaptive compiler.

[7] In this study we used $C = 10$.

6.1 Experimental Setup

The experimental is a 15-node cluster, each featuring two Quad-Core AMD Opteron processors (model 2350) clocked at 2 GHz, with 8 GiB of RAM and 20 GiB of swap space. The CentOS GNU/Linux version 5.2 operating system was used. All experiments use a development version of Testarossa using Java 6 class libraries.

6.2 Model Training

Five models were trained with the data collected from the SPECjvm98 suite. Each benchmark that did not successfully complete the original data collection process was removed due to insufficient data generated.[8] One of the applications in SPECjvm98, *_200_check*, is not included in the experiments since it is not really a benchmark. It is only intended to check whether the JVM is able to execute the benchmark suite.

Table 1 summarizes the models created. Each row represents a separate experiment with a specific testing set and non-intersecting testing set. The Model column identifies which benchmarks contributed their data to the training dataset. The models are identified with the initials of the benchmarks used to create a training dataset: `co` for *_201_compress*, `db` for *_209_db*, `mp` for *_222_mpegaudio*, `mt` for *_227_mtrt*, and `rt` for *_205_raytrace*. The columns within Collected-data column characterize the data generated with the benchmarks. For each row in the table, the total number of collected data points is displayed in the Data Instances column. The data points are distributed over the number of data classes shown in the Unique-classes column. The number of distinct feature vectors is presented in the Unique-feature-vectors column. The Ratio column presents the average ratio of data instances for each unique class.

The Ranked-data columns depict the result of the ranking process over the input data. For each unique feature vector, the 3 best ranked unique compilation plans are selected for inclusion in the training set, provided their ranking value is at least 95% of the best one. These ranking parameters greatly reduce the number of data instances selected for training, which is presented in the Training-instances column. The amount of data instances selected for training ranges from 0.06% to 0.07%. In the Training-classes column, the number of classes selected for training is significantly smaller as well, ranging from 2.26% to 2.94% of

[8] The data collection process creates compilation plans that were not anticipated for offi cial releases of the compiler, and thus exposes unknown dependencies between transformations. While uncovering such dependencies is valuable for the compiler engineering team, they prevented the generation of a larger training set.

Table 1. Summary of the data collected as it is processed until learning a model.

Data Set	Collected Data				Ranked Data				
	Data Instances	Unique Classes	Unique Feature Vectors	Ratio	Training Instances	(% original)	Training Classes	(% original)	Ratio
co db mp mt	710,317	51,489	1,200	1:13.80	2,323	0.07	1,428	2.77	1:1.63
co db mp rt	690,072	51,470	1,178	1:13.41	2,302	0.07	1,380	2.68	1:1.67
co db mt rt	583,775	51,490	1,000	1:11.34	1,884	0.06	1,164	2.26	1:1.62
co mp mt rt	714,721	48,007	1,155	1:14.89	2,254	0.06	1,400	2.92	1:1.61
db mp mt rt	751,567	48,037	1,148	1:15.65	2,224	0.06	1,413	2.94	1:1.57

the amount before the ranking process. Finally, the Ratio column displays the average ratio of training instances for each training class in the dataset.

The selection of only a handful of compilation plans for each unique feature vector and the enforcement of a samples-to-class ratio reduced the data used to train the LIBLINEAR SVMs. After extensive tuning, the training times reduced from several hours (days, in some cases) down to a few minutes. Each of the models presented in this study was trained in approximately 2 minutes.

6.3 Experimental Methodology

The five models shown in the rows of Table 1 were generated to enable leave-one-out cross-validation amongst the five benchmarks for which the collection of training data completed. The results of this cross-validation are shown in the columns for *raytrace*, *mtrt*, *mpegaudio*, *db*, and *compress* respectively in Figure 3. The remaining benchmarks can be considered reserved sets for testing as their data was never used in training.

All benchmarks are executed using the large inputs of the suite. For the throughput evaluation, the benchmarks are internally iterated 10 times to amortize JVM startup latency. In the start-up evaluation the benchmarks execute for only one iteration. Each measurement is repeated 30 times to account for internal (*e.g.*: garbage collection and set of methods compiled) and external (*e.g.*: thread scheduling by the operating system and I/O latencies) factors. Figures 3 and 4 show the average and the standard deviation for the measurements. The overhead of the communication with the model is usually less than 0.1% of the running time (depending on the number of compilations performed and the size of the model), averaging 214 µs per request.

6.4 Experimental Results

Figure 3 compares the relative throughput performance of Testarossa using the machine-learned models with the unmodified compiler, considering 10 internal benchmark iterations. For each set of benchmarks, the leftmost bar is the

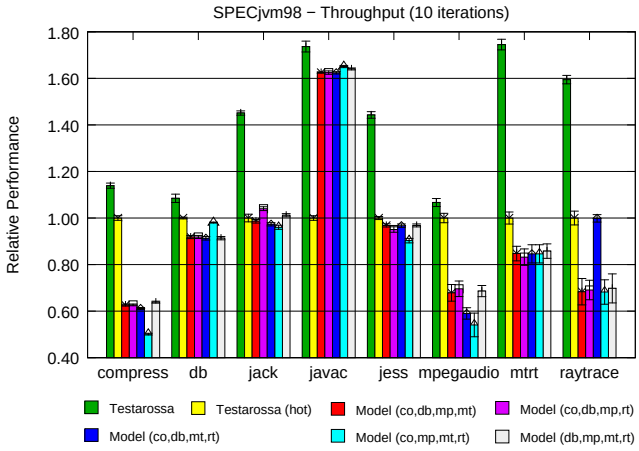

Figure 3. Relative throughput performance based on Testarossa using hot strategy for 10 iterations of SPECjvm98 benchmarks.

relative performance of the unmodified Testarossa under its regular adaptive operation. The second bar is the hot compiler that is forced to only use hot compilation plans which were not modified by the learned models. The remaining five bars contain the relative performance for each learned model. In these results we consider the time to complete all 10 iterations of each benchmark measured.

In the results for *compress*, *db*, *mpegaudio*, *mtrt*, and *raytrace* only the bars for models that do not include the benchmark being tested in the training can be used for cross-validation. For instance, for *db* only the column corresponding to the model `co,mp,mt,rt` should be considered. However, the measurements for the other models, which include the tested benchmark in the training, provide evidence that, for these five benchmarks, there is little, if any, change to performance when the tested benchmark is included in the training.

The original Testarossa that adaptively selects the optimization level, but applies a pre-determined compilation plan for all methods at a given level, consistently outper-

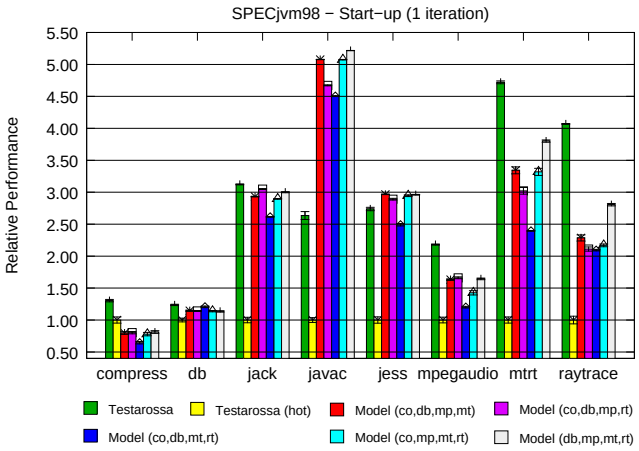

Figure 4. Relative start-up performance based on Testarossa using hot strategy for the first iteration of SPECjvm98 benchmarks.

forms all other versions of the compiler across all benchmarks. A reasonable explanation for the superior performance of the original Testarossa, which was hand-tuned over many years, is that the execution times of several of these benchmarks are dominated by a set of methods that must be compiled at the highest optimization level as soon as possible, while keeping the compilation of less-frequent methods at lower optimization levels.

The *javac* performance using the machine-learned models is comparable to Testarossa under adaptive operation. This result provides interesting evidence that the machine-learned models were able to tailor the hot compilation plan on a per-method basis. The *javac* benchmark is the most realistic of the benchmarks in the set because it performs a common important task: compiling Java programs.

Figure 4 presents the relative start-up performance. For start-up execution the machine-learning-based compiler outperforms the hot compiler in most benchmarks (except for *compress*). This result is not surprising because Testarossa is not designed to optimize start-up performance. For *javac*, and most of the models for *jess*, the compiler using the machine-learned model outperforms Testarossa under adaptive operation.

The main reason for the performance of the compiler using the machine-learned models at the hot optimization level being comparable to Testarossa under adaptive operation is that the compilation times are significantly smaller when using the machine-learned models. In the case of *javac*, the savings in compilation time when using the machine-learned models are significant enough for the benchmark to complete its first iteration in about half the time used by Testarossa under adaptive operation, and about 20% of the time used by Testarossa using only the hot optimization level.

7 Related Work

Eeckout *et al.* [7] propose an automated compilation-plan tuning based on multi-objective evolutionary search that uses Jikes as a testbed. In their approach, Jikes can be fine-tuned for different (or mixed) scenarios: a specific hardware platform, a set of applications, or a set of inputs for applications of interest. The tuning is carried out as a two-step process, starting with the exploration of different compilation plans to identify those that are Pareto-optimal, and then assigning a subset of them to the JiT during a fine-tuning step. Compilation plans are ranked in terms of their code-quality output and compilation rate. Pareto-optimal plans are those that yield the best-performing code and compilation rate within a set of neighboring plans. The Pareto-optimal plans form a Pareto frontier, restricting the number of compilation plans evaluated during the fine-tuning step. In the fine-tuning step, the Pareto-optimal plans are evaluated considering all effects present in the JiT compiler (*e.g.*: GC activity, which was avoided in the exploratory step to allow convergence of the search algorithm) and the adaptive compilation system, so that the final result is consistent with the expected use in a JVM.

The work most closely related to this paper is an interesting variation on this theme: to learn a particular set of code transformations that should be applied to each given method of a program. Cavazos and O'Boyle [2] trained machine-learning models based on logistic regression to work with the Jikes RVM (Research Virtual Machine), a multi-level adaptive JiT Java compiler that does not interpret Java bytecode when executing an application. They use a set of 26 features to describe methods in the form of counters (*e.g.*: length of the method in Java bytecodes), attributes, and distribution of Java bytecodes. In addition, three models are trained, one for each optimization level. For the lower optimization levels (`-O0` and `-O1`), data is collected for all possible permutations, respectively 16 and 512 compilation plans. As the transformation space for `-O2` would be impractical to exhaust (there are 2^{20} possible plans), they collect data for 1000 randomly generated plans. The training datasets are created by ranking data samples on a per-method basis, selecting those samples within 1% of the best performing method-specific plan. They report improvements both on compilation and running time for the fixed scenarios, *i.e.*, when the compiler is set to compile methods at a specific optimization level (`-O0`, `-O1`, and `-O2`). However, they have limited success when comparing with the adaptive strategy in the Jikes compiler. These results are consistent with the our preliminary findings in Testarossa.

Traditionally the application of supervised learning methods to compilation have used a set of program features

specified by compiler developers based on their intuition and experience. One limitation of this methodology is that the learning might be biased towards what compiler designers already know about the program's behaviour. An interesting alternative was proposed recently by Leather *et al.*: to create a system that can automatically generate features for the learning algorithm based on performance measurements of the code [12].

A compiler must determine not only *which* transformations to apply to a program, but also the *ordering* in which the selected transformations should be applied. The ordering is important because a transformation A may either create or eliminate an opportunity to apply a transformation B. Early studies indicate that the search space is extremely complex and unyielding — thus limiting the applicable learning to variations of random search, such as genetic-based techniques [1]. Later studies showed that pruning can result in higher yielding search spaces [10, 11]. A limitation of this application of learning is the difficult integration of phase reordering into a commercial compiler. The research is based on academic compilers that were specifically developed by a small group to allow reordering of phases. Engineering this application of learning into a commercial compiler is challenging. Unlike the academic compilers where phase ordering have been studied, commercial compilers have large code bases that are developed over many years by dozens, or even hundreds, of developers. Therefore, reordering transformations may require significant engineering and testing efforts.

8 Conclusion

This paper describes several challenges that need to be overcome to enable the use of SVMs to learn method-specific compilation plans in a commercial-grade JiT compiler. It describes in detail the data collection process and the training of the SVM model and summarizes preliminary results from an extensive and thorough experimental evaluation of the methods described. The results indicate that for an important type of benchmark, the benefit from selecting method-specific plans based on a machine-learned model is comparable with the hand-tuning of the compilation strategies by dozens of expert developers over many years. The results also indicate that the start-up performance of applications can benefit from method-specific compilation.

Copyright and Trademarks

References

[1] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Language, Compiler and Tool Support for Embedded Systems (LCTES)*, pages 231–239, Washington, DC, 2004.

[2] J. Cavazos and M. F. P. O'Boyle. Method-specifi c dynamic compilation using logistic regression. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 229–240, Portland, OR, 2006.

[3] D. Detlefs and O. Agesen. Inlining of virtual methods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278, London, UK, 1999. Springer-Verlag.

[4] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classifi cation. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[5] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java™ Just-In-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[6] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Verlag, 2009.

[7] K. Hoste, A. Georges, and L. Eeckhout. Automated just-in-time compiler tuning. In *Code Generation and Optimization (CGO)*. ACM, 2010.

[8] IBM Corporation. `http://www.ibm.com/developerworks/java/jdk/`.

[9] S. Keerthi, S. Sundararajan, K. Chang, C. Hsieh, and C. Lin. A sequential dual method for large scale multi-class linear SVMs. In *Knowledge Discovery and Data Mining (KDD)*, pages 408–416. ACM, 2008.

[10] P. A. Kulkarni, S. Hines, J. Hiser, D. B. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Programming Language Design and Implementation (PLDI)*, pages 171–182, Washington, DC, June 2004.

[11] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Exhaustive optimization phase order space exploration. In *Code Generation and Optimization (CGO)*, pages 306–318, New York, NY, 2006.

[12] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Code Generation and Optimization (CGO)*, pages 81–91, Seattle, WA, USA, 2009.

[13] Standard Performance Evaluation Corporation (SPEC). `http://www.spec.org/jvm98/`.