Using Machines to Learn Method-Specific Compilation Strategies

Ricardo Nabinger Sanchez
José Nelson Amaral Duane Szafron
University of Alberta,
Edmonton, AB, Canada
{nabinger,amaral,duane}@cs.ualberta.ca

Marius Pirvu Mark Stoodley
IBM Toronto Software Laboratory,
Markham, ON, Canada
{mpirvu,mstoodle}@ca.ibm.com

Abstract

Support Vector Machines (SVMs) are used to discover method-specific compilation strategies in Testarossa, a commercial Just-in-Time (JiT) compiler employed in the IBM® J9 Java™ Virtual Machine. The learning process explores a large number of different compilation strategies to generate the data needed for training models. The trained machine-learned model is integrated with the compiler to predict a compilation plan that balances code quality and compilation effort on a per-method basis. The machine-learned plans outperform the original Testarossa for *start-up performance*, but not for *throughput performance*, for which Testarossa has been highly hand-tuned for many years.

1. Introduction

The success of dynamic programming languages such as Java™ relies heavily on the underlying runtime support provided by interpreters, JiT compilers, and virtual machines (VMs). JiT compilation generates native code that can execute directly on the host platform, while the application is running. Modern JiT compilers implement many code transformations to improve the execution performance of the code generated. However, a balance must be struck between compilation effort and code quality because the compiler competes with the application for the same resources [1, 3]. JiT compilers operate at multiple optimization levels (compilation plans) that are selected in reaction to the execution behavior of the application. A JiT control unit spends its compilation effort in proportion to the expected execution time of different sections of the application. For example, a few key methods in a Java application may be compiled at the highest optimization level, whereas infrequent or short-lived methods may be compiled at the lowest optimization level or may not be compiled at all [2].

Selecting the combination of code transformations that should be included in each compilation plan requires non-trivial effort. The state of practice in industry is that the compilation plan for each optimization level is hand-tuned by compiler experts over many years across a wide range of platforms. When the compiler must support new platforms, existing compilation plans may require adjustments or may need to be completely redesigned. In addition, maintaining the optimization levels is difficult because changes in a compilation plan may improve the performance of some applications while degrading others [19].

O'Boyle and Cavazos have proposed using machine learning to tailor compilation plans on a per-method basis [6]. The idea is to extract features from methods, and to present the machine-learning algorithm with multiple variations of the original compilation plan. The algorithm creates a model that identifies patterns in the data presented and predicts a method-specific compilation plan for unseen methods that have similar features to the ones examined. For instance, the learned model may discover: (i) transformations that lead to slower code and should be disabled for specific methods; (ii) alternative transformations that can deliver equivalent performance at a reduced compilation cost; (iii) combinations of transformations that improve performance.

This paper describes a complete framework that integrates method-specific compilation strategies selected by a machine-learned model in Testarossa, an enterprise-grade, commercial Just-in-Time compiler for Java from IBM used in the IBM J9 Java Virtual Machine. Significant contributions in this paper include:

- A data collection infrastructure that performs compilation experiments using a lightweight method profiling mechanism.
- A customized binary archive format to facilitate large-scale data collection experiments.
- Supporting tools to convert archives into the format required by the SVM implementation.
- A lean and versatile communication protocol that integrates the machine-learned models with the compiler and allows different models to be easily swapped without changes to the compiler.
- An extensive experimental evaluation that shows that the models outperform an out-of-the-box development version of Testarossa for start-up performance, but not for throughput performance. Moreover, the models reduce compilation time by half.

Section 2 introduces the IBM Testarossa compiler. A brief overview of SVM is given in Section 3. Section 4 presents the

CGO 2011 April 02–06, Chamonix, France.

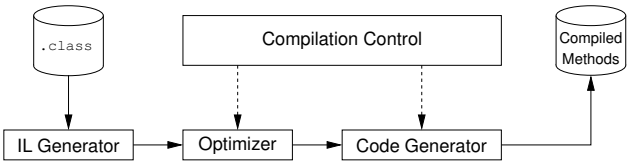

Figure 1. Architecture of IBM Testarossa.

data collection infrastructure. The exploration of different compilation plans is discussed in Section 5. The process of training an SVM model is described in Section 6. The mechanism for integrating the machine-learned model with the compiler is discussed in Section 7. Section 8 reports on the experimental evaluation, and Section 9 discusses related work.

2. IBM Testarossa

JiT compilers can be decomposed into two key components: (*a*) a profiling mechanism that identifies portions of the application that are likely to benefit from JiT compilation; and (*b*) the JiT compiler itself. Profiling mechanisms keep track of the areas in the application that are frequently executed. An example of a profiling mechanism is per-method invocation counters. The Java virtual machine (JVM) uses this profiling information to decide whether it is worth compiling a method to native code, while the JiT compiler also uses the profiling information to decide how to optimize the method.

Testarossa implements many optimizations [13, 17] and features an adaptive compilation strategy with five levels identified by adjectives related to temperature: *cold*, *warm*, *hot*, *very hot*, and *scorching*. The temperature estimates how frequently a method is executed.[1] The hotter a method is, the more compilation effort Testarossa invests in it. Testarossa uses a combination of invocation counters and time sampling to estimate the hotness of a method. The goal is to anticipate the compilation of methods that spend a significant amount of time during fewer invocations. A method is also recompiled at a higher optimization level if it executes frequently.

Figure 1 illustrates the four major components in the Testarossa architecture. The **IL Generator** converts Java bytecodes loaded from Java class files into a tree-form intermediate language (IL) that is used as both input and output during the optimization process. The **Optimizer** performs code transformations on the IL-tree, and the final tree is fed to the **Code Generator**. The code generator translates the IL-tree into native instructions for the supported platforms (*e.g.*: Intel x86, MIPS, PowerPC®, S/390®, and others). The **Compilation Control** decides when to compile (or recompile) a method and which optimization level should be used. These decisions are based on dynamic execution profiles. Optimization consumes most of the compilation resources.

Each optimization level has an ordered set of code transformations (a *compilation plan*) that are applied on the IL-tree of a method. A plan may apply from 20 transformations (*cold*) to more than 170 (*scorching*), including the multiple application of some transformations that are used as cleanup steps. Before applying a transformation prescribed by a plan, the compiler checks for method characteristics that might make the transformation meaningless. For instance, loop transformations are never applied to methods that do not contain loops. Unless a rare dependence requires synchronization with the execution of the application, compilations are performed asynchronously in separate threads.

[1] In the jargon of developers, instead of talking about the temperature of a method, one talks about the *hotness* of the method.

3. Support Vector Machines

Support Vector Machines (SVMs) are statistical learning models that work by finding maximum separating hyperplanes in a space formed by data instances (each represented by a p-dimensional vector $\vec{X}$) and associated classes (sometimes called labels) [7, 14]. The trained SVM is used to predict the class of an unseen $\vec{X}$ by computing the location of $\vec{X}$ relative to separating hyperplanes.

One key advantage of SVMs is their flexibility. They can handle overlapping data by allowing some level of misclassification in order to optimally place a separating hyperplane. The maximization of the separating margins provide a high generalization power even with a small set of training instances. An adjustable parameter, called misclassification cost C, allows the training to be tuned by making the model more or less flexible while preventing the overfitting of the resulting model.

The experiments described in this paper use the multiclass classifier implementation in LIBLINEAR [9], which uses a version of SVM [18]. This version performs well on large-scale classification problems that have numerous instances and/or distinct classes. The learned model consists of a $p \times L$ matrix containing real valued weights that represent the contributions of each of the p features used to separate the distinct classes. The prediction time is proportional to the size of the matrix.

4. Data Collection

The data-collection process mimics the regular operation of Testarossa, but performs controlled changes in the compilation plans to explore alternative scenarios on a per-method basis. The data collection starts when the VM selects a method for compilation (Figure 2 (a)). Method features are computed and recorded before the compilation starts. A compilation plan is chosen based on the compilation level selected by the VM (b). A compilation-plan modifier is retrieved from a queue of pre-computed modifiers (c) and combined with the original compilation plan, instructing the JiT to perform a different set of code transformations (d). As is the case with most production compilers, the order of transformations cannot be easily changed without violating dependencies among them (which can either cause compiler malfunction or generation of invalid code); thus transformations may be removed from the original compilation plan but no transformations are added and transformations are not reordered. The freshly compiled method is inserted in the pool of compiled methods (e). The JiT may select a method to recompile at a higher optimization level (f). The JiT compilation also instruments the method to collect dynamic information, such as execution time, at each invocation. For data collection, recompilation requests to the JiT are generated after a fixed number of invocations of a method. Thus, many compilation-plan modifiers are explored in a single JVM execution.

We added a *strategy control* component to Testarossa to control the exploration of modifiers during data collection. A modifier is retired after it has been used for a certain number

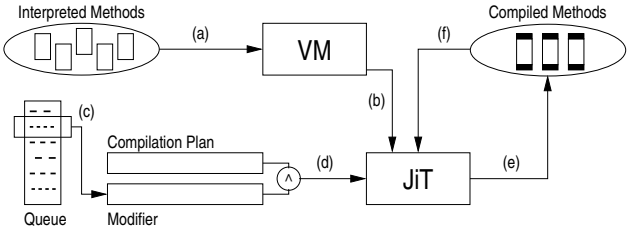

Figure 2. Data Collection during training (from [22]).

Table 1. Scalar features.

Counters	Attributes	
Exception handlers Arguments Temporaries Tree nodes	Constructor? Final? Protected? Public? Static? Synchronized?	Allocates dynamic memory? Unsafe symbols? Uses BigDecimal? Virtual method overridden?
	Many-iteration loops? May have loops? May have many-iteration loops?	Strict floating-point? Uses floating-point?

of compilations. The method is never compiled twice with the same modifier. A special *null modifier*, which does not change the original Testarossa compilation plan, is tried with every compiled method to ensure that the machine-learned model will be exposed to the original compilation strategy in the compiler.

4.1 Method Features

A feature vector $\vec{F}$ is used to characterize a method. $\vec{F}$ should contain enough information about a method to allow a machine-learned model to correlate this information to a compilation plan that yields good performance. To be consistent with data collection, $\vec{F}$ is dynamically extracted from the compiler just prior to the optimization stage.

In this implementation, $\vec{F}$ is composed of 71 numerical attributes (dimensions). These attributes can be organized in two sets: **scalar features** consist of counters and binary attributes for a given method without any special relationship; and **distributions** characterize the actual code of the method by discriminating between operand types that appear in the method and by aggregating similar operations. The set of features is a selection of measurements already available in Testarossa. This selection is based on developer's intuition. The original set of features was gradually reduced as data collection provided evidence that some of the features were invariant across all applications used for data collection.

4.1.1 Scalar Features

Table 1 illustrates how the scalar features in the feature vector $\vec{F}$ are grouped as *counters* and *attributes*. The *exception handlers* counter indicates how many exception handlers are present in the method. *Arguments* and *temporaries* partition the set of all symbols referenced in the method into two disjoint sets. *Tree nodes* is the number of nodes used to represent the method in the intermediate language used by Testarossa.

Table 2. Type-based distribution features.

Java Native			Testarossa	Learning-only
Scalar		Non-scalar		
`byte`	`long`	Address	Long double	Mixed types
`char`	`float`	Object	Packed decimal	
`short`	`double`		Zoned decimal	
`int`	`void`			

All attributes are binary (marked with a ? in Table 1). The values of the first subgroup of binary attributes are explicit in the method (*constructor*) or denoted by keywords (*final*, *protected*, *public*, *static*, *synchronized*. The next subgroup is related to the presence of loops: *may have loop* is true if the method has a backward branch; the values of *many-iteration loops* and *may have many-iteration loops* are based on loop-count thresholds and on the presence of nested loops.

The next subgroup contains a diversified set of method characteristics. *Allocates dynamic memory* triggers specific passes, such as escape analysis. *Unsafe symbols* is true for methods that inline a method from the class `sun.misc.Unsafe`, which prevents some optimizations such as redundant-load elimination. *Uses BigDecimal* indicates that arbitrary precision computations use the core Java library `java.math.BigDecimal`. Such computations may not be eligible for rematerialization because the code generated outweighs the benefits of this optimization.[2] *Virtual method overridden* indicates that the method has to be recompiled because it was overridden through dynamic class loading.

The final subgroup records the use of floating-point computation (*Uses floating-point* attribute) and whether the JVM should enforce strict floating-point compliance.

4.1.2 Distributions

Distributions are recorded by incrementing counters until they reach their maximum capacity. There are two kinds of distributions: (*i*) distribution over types (16-bit counters) and (*ii*) distribution over operations (8-bit counters). This separation allows for (*a*) a simpler implementation for the collection process, (*b*) a reduced set of features, and (*c*) a smaller storage requirement. Table 2 presents all of the 14 types counted. Besides the Java native types, distributions also count addresses (an array with one or more dimensions) and user-defined objects [12]. Testarossa supports specialized types such as long double, a quadruple-precision 128-bit IEEE-754 floating-point type, packed and zoned decimals to support Binary-Coded-Decimal (BCD) representations. BCD is used in financial applications and enables fixed-point, arbitrary-precision computation.

Table 3 presents the 38 operations characterized by distributions grouped by type of operation. In addition to instructions required by JVMs, Testarossa supports `compare` and type casts for additional types (*e.g.*: `longdouble`, `packed` and `zoned` decimals). Specialized load/store operations are coalesced into either load, load const, or store operations. To distinguish these operations, each type-specialized form triggers a different type counter. In the *JVM* group, `instanceof` counts the number of tests to verify whether an object reference is of a given type; `synchronization` counts the

[2] *Rematerialization* is a code transformation where the compiler emits code that recomputes a value to reduce register pressure or eliminate loads.

Table 3. Operation-based distribution features.

ALU	Cast	Load/Store	JVM
add	byte	load	instanceof
sub	char	loadconst	synchronization
mul	short	store	throw
div	int		
rem	long	**Memory**	**Branch**
neg	float		
shift	double	new	branch
or	*longdouble*	new array	call
and	address	new multiarray	
xor	object		
inc	*packed*		***Array operations***
compare	*zoned*		***Mixed operations***
	check		

number of serializing Java instructions (`monitorenter` and `monitorexit` instructions); and `throw` reflects the number of `athrow` instructions used to raise exceptions. *Array operations* records array-specific operations — such as bounds check, array copy, and array comparison — in the intermediate representation. A modification of an element in an array is not an array operation because it loads an element (a `load`), performs a computation (an ALU operation), and stores the result (a `store`).

The 52 distribution counters are computed in a single pass over the tree-based representation of a method in Testarossa, just prior to the start of the optimization stage.

4.2 Instrumentation of Methods

The data-collection process collects the time spent compiling each method and the execution time of each method. This time is collected by calls to `TR_jitPTTMethodEnter` and `TR_jitPTTMethodExit`. The exit call is inserted at any exit point and at any block that throws an exception.

The high-resolution time measurements uses the x86 architecture Time-Stamp Counter (TSC), a 64-bit unsigned integer counter that is incremented at every CPU clock cycle. The instructions that read this counter (`rdtscp`[3]) also records the processor identifier. Checking that the identifier is the same in the enter and exit measurements for a method, and discarding the measurement when they are not, avoids the type of imprecision caused by *TSC drift*, a frequent condition where two cores operate at slightly different frequencies.[4] On Linux, the load balancer migrates threads roughly once every 200 ms [20]. In practice, load balancing occurs once every few seconds (≤ 10 s).

The instrumentation also sends recompilation requests to the JiT compiler when the number of executions of the current version of the methods reaches a threshold. This recompilation threshold: depends on the execution time of the method; is computed during the first eight invocations of the method when it is compiled for the first time; and can vary between 50 and $50,000$. The goal is for the method to accumulate the equivalent of 10 ms of running time between compilations.

The data gathered in collection mode is stored in carefully designed data structures in memory and is only transferred to compact binary archives after the execution of the application terminates. Designing a compact representation for the data gathered was crucial because additional I/O operations during the execution of the application would interfere with the dynamic execution and with JiT compilations. The creation of a dictionary of method signatures is key for a compact representation of the data collected [21]. The customized infrastructure for data collection features low overhead, compact storage and high-resolution.

[3] Stands for *read time stamp counter with processor identifier*.

[4] The alternative, synchronizing the TSC across multiple cores, has significant overhead.

5. Compilation-plan Modifiers

A compilation-plan modifier is a sequence of bits. Each bit determines whether a code transformation is enabled. Modifiers are used both during data collection and when the machine-learned model is used during execution. Testarossa's **strategy control** module uses a given modifier to disable transformations. A modifier does not change the order in which the transformations are applied. Two methods are used to generate compilation-plan modifiers: (*i*) a pure randomized search with aggressive exploration; and (*ii*) a progressive randomized search that gradually diverges from the original optimization plan used by Testarossa.

For randomized search, M random modifiers are generated ahead of time for each optimization level. Each modifier is used for 50 compilations (of different methods) and is then expired. This slow rate of exploration allows many methods to be compiled with the same plan and is adequate given that there is significant variation between modifiers. The third modifier used is always the null modifier, which does not change the original Testarossa compilation plan.

The progressive randomized search adds a strong controlled bias to the exploration. The idea is to start with the null modifier and then progressively increase the chance that a given transformation is disabled. The probability that a given transformation will be disabled in the i-th modifier is given by:

$$D_i = i \times \frac{0.25}{L}, \quad 0 \leq i \leq L. \tag{1}$$

where L is the number of modifiers generated for a given compilation level. Thus the probability that each individual transformation is disabled evolves from $D_0 = 0$ to $D_L = 0.25$ as the data collection progresses. The upper limit of 0.25 balances the generation of plans that are increasingly dissimilar to the original plans in Testarossa, but not so different as to be equivalent to plans generated using the randomized approach.

L was experimentally set to 2000 to generate enough modifiers for all the applications submitted to data collection with only a small excess of modifiers. If an application executes long enough for a method to be recompiled more than L times, that method is no longer recompiled while still allowing other methods to be recompiled. If all methods eligible to be compiled reach L recompilations, the data collection is gracefully terminated, and the application is allowed to execute normally, without further recompilations.

The increase rate of 0.000125 per round makes the search gradually diverge from the original compilation plan in Testarossa. Thus the search for alternative compilation plans is likely to be concentrated over plans similar to the original ones in Testarossa. The premise is that the compilation plans included in Testarossa should be good for most methods in most applications, with some applications requiring small modifications to some specific methods. In practice, modifiers that diverge too much from the original compilation plan (*e.g.*: 50% or more transformations disabled) tend to exhibit poor performance.

Theoretically, *each* distinct feature vector representing a distinct method[5] has a space of 2^L possible modifiers to explore. In this implementation, there are 58 distinct code transformations that are controllable, leading to a search space of $2^{58} \approx 2.88 \times 10^{17}$. In practice, most of these modifiers will not create better optimization plans. Thus a heuristic-based search that evaluates the performance for modifiers during data collection may focus the search on promising regions within the space of possible modifiers. The implementation of such a search is left for future work.

6. Learning a Model

The training of a machine-learned model involves processing the data collected and selecting appropriate parameters for the SVM. The data processing involves: (*i*) unarchiving collected data into intermediate data sets; (*ii*) optional merging of intermediate data sets; (*iii*) ranking the data; (*iv*) adjusting the data set to meet training requirements. For example, the data set size must match the resources available for training. This balance is also achieved by selecting parameters for the SVM, such as the misclassification cost and the kernel function to be used.

Unarchiving extracts information from the compact archives and stores it in a format that is suitable for further processing. Merging of intermediate data sets allows for the selective use of data sets of interest to enable *cross-validation* and *leave-one-out cross-validation*. The following function is used to compute the ranking of the i-th record in a data set:

$$V_i = \frac{R_i}{I_i} + \frac{C_i}{T_h} \tag{2}$$

where R_i is the accumulated running time of the method compiled using the respective modifier, I_i is the invocation counter, C_i is the compilation time, and T_h is the triggering value used by Testarossa for recompiling at compilation level h (h is used to reflect the *hotness* or optimization level).[6] Thus, V_i is the normalized cost for the i-th record, combining the average time spent in a single invocation of the method and the compilation cost normalized based on the expected behavior of the loops in the method (if any). Compilation plans that have smaller V_i are better. After ranking, each unique feature vector has a set of pairs $\langle M_i, V_i \rangle$, where M_i is a modifier and V_i is the value of the ranking function for that modifier when applied to a method with the feature vector. There will be at least one such pair for each unique feature vector.

Figure 3 illustrates the ranking work flow, which is very simple but CPU-intensive. Intermediate data sets are loaded and progressively sorted in lexicographical order, based on the feature vector of each record. This sorting aggregates all experiments performed on the same feature vector. Three alternative strategies are used to select the set of modifiers that will be used to train the model for each unique feature vector: (i) use only the best modifier; (ii) use the top N modifiers; (iii) use the top M% best modifiers.

[5] In this study, methods are as distinct as their respective feature vectors.

[6] For each optimization level, Testarossa uses three distinct compilation triggers: one for methods without loops, a second one for methods likely to have loops, and a third one for methods containing many-iteration loops. The default setting in Testarossa is to compile methods that contain loops sooner than those methods without loops, and even sooner if the method may contain many-iteration loops.

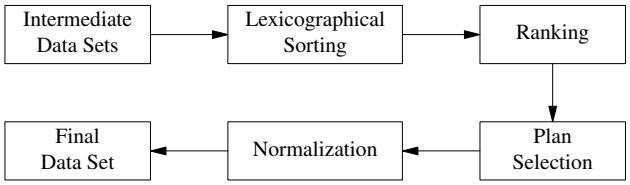

Figure 3. Processing of intermediate data sets

Class label	Feature vector components			
L_i	$\texttt{1:}F_{i,1}$	$\texttt{2:}F_{i,2}$	...	$\texttt{m:}F_{i,m}$
L_{i+1}	$\texttt{1:}F_{i+1,1}$	$\texttt{2:}F_{i+1,2}$	...	$\texttt{m:}F_{i+1,m}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
L_n	$\texttt{1:}F_{n,1}$	$\texttt{2:}F_{n,2}$	...	$\texttt{m:}F_{n,m}$

Figure 4. Data set format used by LIBLINEAR

Each component of each feature vector is normalized to the $[0, 1]$ range using

$$C_{\text{norm}} = \frac{C_j - C_{\min}}{\Delta C} \quad 0 \le C_{\text{norm}} \le 1, \tag{3}$$

where C_j is the j-th component of the feature vector, $C_{\min}$ the minimum value seen during data processing, and ΔC the difference $C_{\max} - C_{\min}$ (where $C_{\max}$ is the maximum value of the j-th component). This normalization eliminates the dominant effect of larger numerical ranges over smaller ones when an SVM is trained [18].

The format of the final data obeys the rules required by LIBLINEAR. The data sets use a textual sparse-matrix format, where each line is a data instance used as input when training the machine-learned model.

Figure 4 illustrates the format symbolically. The data set is composed of n data instances, each on a single line of the file. Each i-th data instance is described starting with the respective class label, followed by m components of the feature vector. Features with value zero can be omitted and all non-zero components must be preceded by its component index. For example, `10:0.5625` indicates that the 10-th component of the feature vector has value `0.5625`. The LIBLINEAR implementation requires class labels in the $[1, 2^{31} - 1]$ range. Thus, the compilation plan modifier space is remapped into this smaller space.

In order to train a machine-learned model based on SVM, two parameters must be defined: (a) the weight C used by the model to deal with (inevitable) misclassifications, and (b) the SVM kernel that will be used. In this study, a value of $C = 10$ was empirically selected to balance the quality of the model generated and the training time. When selecting a kernel, there is a trade-off between the ability of the model to adapt to correlations within the training data and its capability to both train and perform predictions in a timely fashion. There are two factors to consider in kernel selection: (a) dimensionality of the feature space, and (b) time budgets, consisting of the time available for training a model, and the time constraints when performing a prediction. This study discovered experimentally that the non-linear kernel radial basis function (RBF) had low training times (around 20% of the training time of the linear model), but its prediction speed was very low — a learned RBF model can take up to

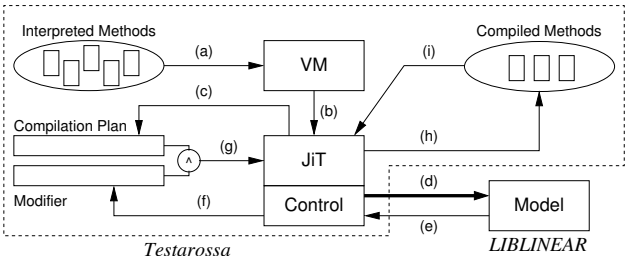

Figure 5. Learning-Enabled Testarossa.Architecture

660 ms to compute a prediction.[7] It should not take longer to find out which transformations to apply to a method than to compile that method at the highest optimization level.[8] On the other hand, a linear kernel leads to longer training time, but the time to compute a prediction can be as low as 48 µs (4 orders of magnitude faster than the non-linear kernel).

7. Integrating Compiler and Model

The operation of learning-enabled Testarossa, whose architecture is shown in Figure 5, is similar to the data collection process. The initial decision to compile a method remains unchanged from the current Testarossa (*a*). When such decision is made, the VM instructs the JiT to compile the chosen method (*b*). Next, the JiT selects the appropriate optimization level (*c*). When the compilation is about to start the optimization stage, the Strategy Control extension computes the features for the method being compiled and transmits them to the learned model (*d*) (the thicker arrow indicates that the Control unit provides more information to the model). The model computes a compilation plan modifier and sends it to the Control extension (*e*). Control installs the compilation-plan modifier (*f*) to disable some of the transformations that would otherwise be applied (*g*). After the compilation, the newly compiled method is installed in the pool of compiled methods (*h*). As the application executes, the JiT eventually decides to recompile methods (*i*), repeating the process.

Several practical issues must be dealt with in the implementation. Each feature vector has to be renormalized using the same parameters that were used for normalization in the data collection (the shift and scale parameters are saved in a *scaling* file). The output of the machine-learned model is in the $[1, 2^{31} - 1]$ range and has to be mapped back to the full binary pattern that represents a modifier before the model responds to Testarossa. This mapping is done using a lookup table that associates known identifiers in the model with a compilation plan modifier, loaded during the initialization of the model.

The machine-learned model is in a separate process and the communication between Testarossa and the model uses *named pipes* [23]. This approach leads to a flexible prototype enabling the machine-learned model to be replaced without any change to the rest of the infrastructure. The disadvantage is that, compared to a dynamic library, named pipes introduce some overhead. However, we found that, in practice, this overhead is negligible.

[7] The time spent computing a prediction is platform-dependent.

[8] Compilation time is method dependent, but in the experimental platform used in this study, Testarossa can compile many methods in the highest optimization level in about 100 ms to 220 ms.

8. Experimental Evaluation

This evaluation compares the start-up performance, throughput performance, and compilation time of the machine-learned compiler with the unmodified Testarossa compiler.

8.1 Experimental Setup and Methodology

Data collection and performance measurements use a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors (model 2350), with 8 GB of RAM and 20 GB of swap space, running CentOS GNU/Linux version 5.2. No other applications are running. Each JVM invocation was run 30 times to account for disturbances (*e.g.*: scheduling policies in the operating system, garbage collection in the JVM), and a 95% confidence interval is presented along with the average of each measurement. A JVM invocation is the complete execution of an application. Some benchmarks allow for an arbitrary amount of internal iterations within a single JVM invocation to reduce measurement noise caused by start-up effects. A development version of Testarossa is used for all measurements.

Separate models are trained for three optimization levels (*cold*, *warm*, *hot*). The model used for a given compilation is based on the level selected by Testarossa's heuristics. The *scorching* level uses its own instrumentation of methods to enable feedback-directed optimization and that instrumentation conflicts with the instrumentation used to collect data for learning. Therefore, a learned model was not generated for *scorching*. When Testarossa selects scorching, the original compilation plan is used. The evaluation uses the SPECjvm98 benchmark suite and most benchmarks from DaCapo 9.12 [4] with their largest input.[9]

The collection of data for training stresses the compiler in ways that are not supported by the development team. Therefore, data collection was limited to five SPECjvm98 benchmarks that successfully compiled with all the modified compilation plans and whose outputs matched the expected outputs. To enable leave-one-out cross-validation, five sets of models were trained with the SVM, each including four benchmarks. Each set consists of three models, one for each optimization level except the scorching level. In the graphs, two letters identify each benchmark included in a model as follows: `co` for *_201_compress*, `db` for *_209_db*, `mp` for *_222_mpegaudio*, `mt` for *_227_mtrt*, and `rt` for *_205_raytrace*. Data generated (*i.e.*, compilation-plan modifiers) in a session that crashed is not included in the training data sets.

Table 4 presents the average size of the data sets obtained through data collection and used for training for each level of compilation. The training data merges the data from the randomized search and the progressive randomized search data collections. Separate models for each search strategy were also trained and measured, but they did not perform as well as the models that combine both strategies. The Data-Instances column is the size of the training data set. The number of unique classes indicates the diversity of compilation-plan modifiers used for data collection. The number of unique feature vectors represents the number of distinct methods seen by the machine-learning algorithm.

[9] Benchmarks *tradebeans* and *tradesoap* had to be excluded because of errors related to the execution environment, regardless of the compiler used.

Table 4. Average data set sizes used for training the machine-learned models.

	Merged Data				Ranked Data			
Compilation Level	**Data Instances**	**Unique Classes**	**Unique Feature Vectors**	**Vector:Instance Ratio**	**Training Instances**	**Training Classes**	**Training Feature Vectors**	**Vector:Instance Ratio**
Cold	1,551,545	1,421,717	1,175	1:1,320	2,326	949	1,094	1:2.12
Warm	1,577,157	1,455,947	1,153	1:1,368	2,213	1,590	1,108	1:1.99
Hot	2,543,564	2,229,364	1,201	1:2,118	2,073	1,379	1,069	1:1.94

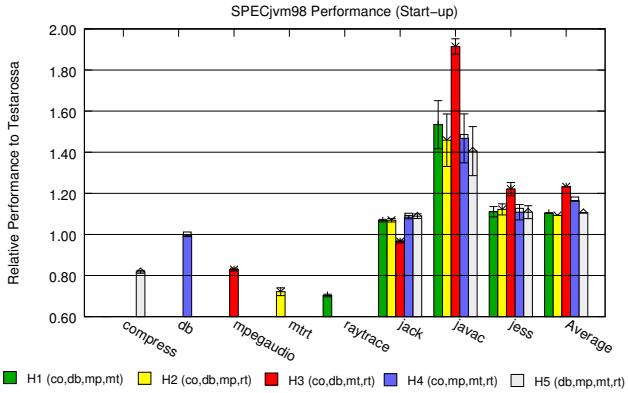

Figure 6. Start-up performance results (single iteration) for SPECjvm98 relative to Testarossa, where higher bars are better.

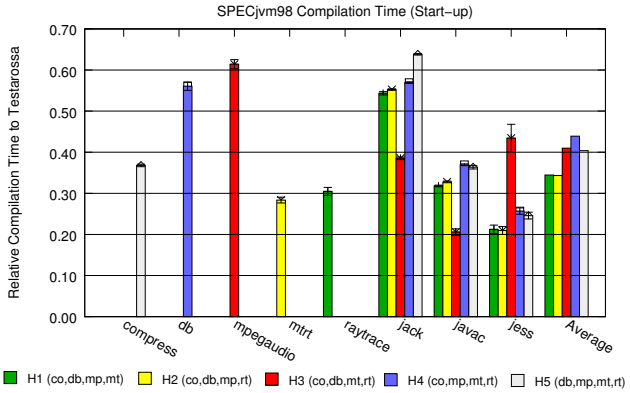

Figure 7. Start-up compilation time (single iteration) for SPECjvm98 relative to Testarossa, where lower bars are better.

Ratio is the ratio of data instances per unique feature vector. The ranking process selects at most 3 compilation-plan modifiers for each unique feature vector. To be selected, a modifier must have a ranking value of at least 95% of the best performing modifier. The size of the data sets are shown in the ranked-data columns. In total, 15 machine-learned models were trained using LIBLINEAR. Each model took 30 to 90 seconds to train with a misclassification cost parameter of $C = 10$.

8.2 Experimental Results

When working with the SPECjvm98, Testarossa developers tune the compiler for *throughput performance*. We define throughput to be the running time of a single invocation of the JVM that repeats the execution of the same benchmark 10 times.[10] This study also measures *start-up performance*, which consists of running a single iteration of each benchmark in each JVM invocation. Start-up performance is more representative of the experience of an actual user of these programs but may not benefit as much from the adaptive compilation system in Testarossa, which is designed mainly for programs with longer running time. For servers, improving start-up performance allows for faster compile-edit-debug cycles, as well as reducing delays where the server is initializing prior to application execution.

Figure 6 presents the start-up performance relative to Testarossa. For benchmarks included in the training set, leave-one-out cross-validation is used — hence the single bar for those benchmarks. The performance for all five models is measured for the benchmarks in the reservation set. Model H3, which leaves out *mpegaudio*, produces significant performance improvement for *javac*. Figure 7 shows that this model also takes much less compilation time. This might be evidence that the learned model can prevent the execution of unproductive code transformations. The variation in the performance gain across the models in the reservation set indicates that the learning process is sensitive to the benchmarks included in the training set. The average performance improvement varies from 10% to 22% while the compilation time is less than half of the compilation time in the unmodified Testarossa. In some instances, such as *jess*, a five-fold reduction in compilation time is observed.

The DaCapo benchmarks are significantly different from the SPECjvm98 benchmarks and thus it is interesting to study the performance of models trained only with benchmarks from SPECjvm98 and evaluated on DaCapo.[11] Positive results will indicate that the SVM-based models are able to generalize for this task. Also, DaCapo benchmarks are not normally used to test the compiler during the development of Testarossa. The relative start-up performance for DaCapo using the five models is shown in Figure 8. Interestingly, the H3 model now has the worse performance and the least reduction in compilation time (see Figure 9). The highest performance gain is measured on *luindex*, a benchmark that indexes a set of documents. The worse is on *h2*, which executes transactions using the model of a banking

[10] Some of the benchmarks run for a short time and therefore the concern is that the process of starting up the JVM may dominate the execution time for a single execution of those programs.

[11] Attempts to generate models using a large number of DaCapo benchmarks failed because the use of unsupported combinations of code transformations resulted in compilation errors.

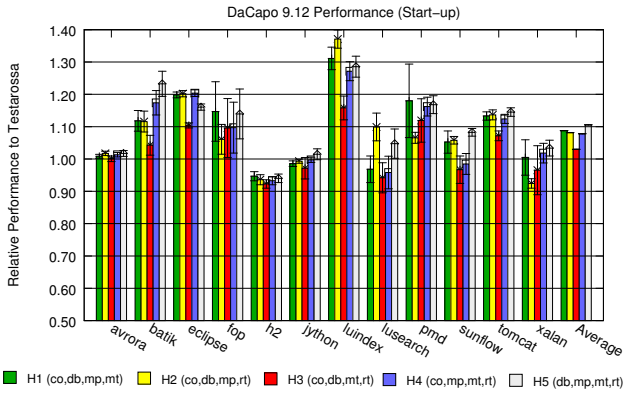

Figure 8. Start-up performance results (single iteration) for DaCapo.

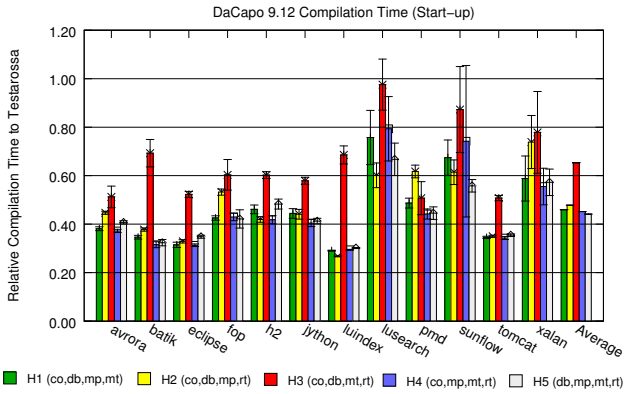

Figure 9. Start-up compilation time (single iteration) for DaCapo.

application. A careful comparison of Figures 8 and 9 indicates a correlation between the performance improvements and the compilation-time reductions, which suggests that the learned models are disabling unproductive transformations. The highlight is that even when presented with a significantly different set of benchmarks, the models delivered a modest performance gain for start-up performance.

The learned models are not as successful when throughput performance is measured as shown in Figures 10 and 11. The comparison is with an unmodified Testarossa that incorporates many years of experience from the development team in the tuning of the compilation plans for each optimization level. When 10 iterations of a (small) benchmark are run within a single JVM invocation, the adaptive heuristic in Testarossa has the opportunity to gather the information that it needs to make good decisions about the compilation level for the benchmark and the compiled methods are run long enough to reap the benefits of the compilation. Still, for a few benchmarks, namely *javac* in the SPECjvm98 and *tomcat* in DaCapo, the trained models outperform the adaptive Testarossa. Moreover, there is less performance variation between models than in the case of start-up performance.

The significant reduction in the compilation time is consistent when throughput performance is measured as shown

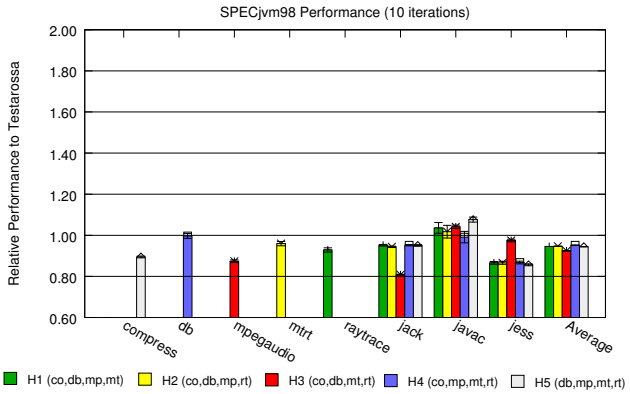

Figure 10. Throughput performance results (10 iterations) for SPECjvm98.

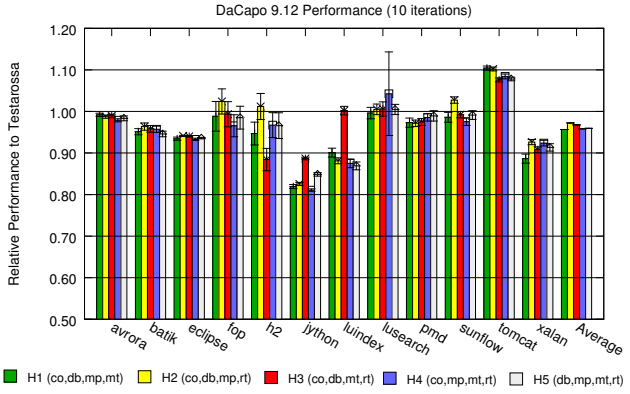

Figure 11. Throughput performance results (10 iterations) for DaCapo.

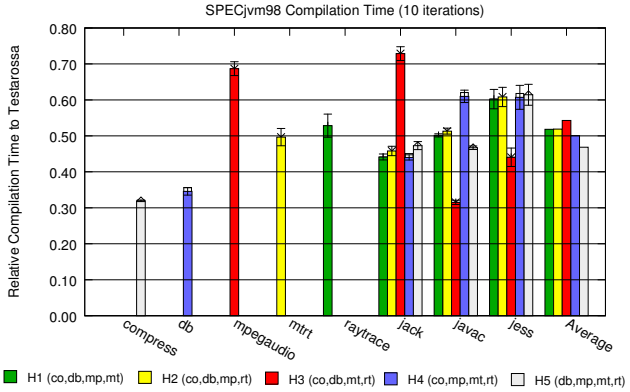

Figure 12. Relative compilation time for SPECjvm98.

in Figures 12 and 13. Again an interesting case is *luindex* where model H3 spends the most time in compilation (reducing it by about 22% in relation to Testarossa) but is the only model to approach the performance of Testarossa. This result indicates that the other models may be disabling code transformations that are actually beneficial to performance.

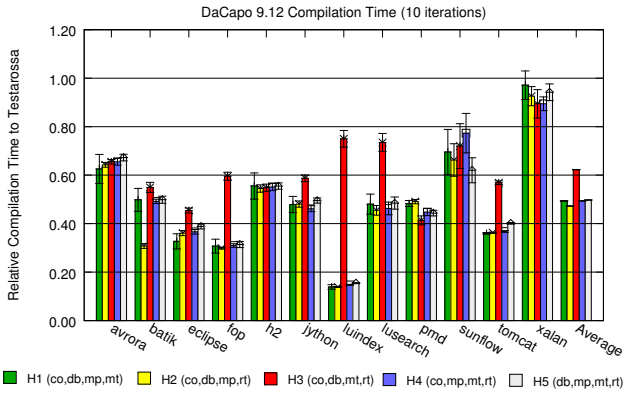

Figure 13. Relative compilation time for DaCapo.

9. Related Work

In the framework that led to MILEPOST GCC, Fursin *et al.* initially focused on evaluating permutations in the optimization space [10]. Later they developed a complete iterative compilation framework that can self-tune on different platforms [11]. The heuristics in the compiler are adjusted for a set of applications on a target platform, depending on the goal of the user: improved execution performance or lower power consumption on embedded platforms, for example. Similar to the approach described in this paper, they fix the order of code transformations, and the machine-learned model is external to the compiler. One key difference is that their models are created on a per-application basis while our models attempt to generalize for multiple applications.

Cavazos *et al.* train a program-specific machine-learning model with a feature space composed of the 60 performance-monitoring events [5]. The learning process applies an unordered set of code transformations to a program used as training input, and samples the performance events. By iterating, the model is exposed to transformations that have a positive or negative impact in the performance counters. When a new, unseen program is fed to the model, it first samples the performance counters to make an initial prediction of which code transformations should be enabled. The process is repeated a few times (depending on the compilation budget set by the user), and the best set of code transformations is applied. Dubach *et al.* use a trained Artificial Neural Network (ANN) to speed up the process of searching for better optimization transformations for a given program [8].

Eeckout *et al.* propose an automated compilation-plan tuning based on multi-objective evolutionary search that uses Jikes as a testbed [16]. The idea is to fine tune the compiler for specific scenarios: a given hardware platform, a set of applications, or a set of inputs for applications of interest. The tuning starts with an exploration of compilation plans to discover those that are Pareto-optimal, and then a subset of those are assigned to the JiT. The goal is to reduce the number of compilation plans evaluated during the second step. Earlier, they had used a similar approach for the GCC compiler [15]. The key difference between this paper and the work of Eeckout *et al.* is that their goal is to adjust the compilation plans in the compiler, which is a coarser approach when compared to method-specific compilation. Their methodology has the advantage of not incurring the overhead of a machine-learned model whenever a compilation is carried out. However, their models need to be retrained to new sets of applications and their plans may not be as fine-tuned for each method as in a method-specific approach.

Vaswani *et al.* use learning to predict the execution time of a program given a set of code transformations, in the GCC compiler, and find that the ideal compilation plan varies from program to program, which is consistent with the findings in this paper [24].

The approach described in this paper uses fine-grained learning in which a model is built to find a good compilation plan for a portion of a program, such as an individual method or a loop. The related work that is closest to our work is by Cavazos and O'Boyle [6]. They trained machine-learned models based on logistic regression to learn compilation strategies for the Jikes Research Virtual Machine (RVM), which includes a multi-level adaptive JiT Java compiler. The models select code transformations to be applied to each method. 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 data sets are created by ranking data samples on a per-method basis and 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.

Their work differs from ours in the following aspects: (i) the number of code transformations available in Jikes is significantly smaller than the number of code transformations in Testarossa (the models in this paper could control 58 code transformations leading to a search space that is several orders of magnitude larger $\log(2^{58}/2^{20}) \approx 11.4$); (ii) in this paper separate models are trained for each optimization level, and Testarossa has an additional optimization level;[12] (iii) the dimensionality of the feature vector in this study is considerably larger; (iv) they use a logistic-regression model, which may output compilation plans not seen during the training, as opposed to a multi-class SVM; (v) Jikes is a research compiler while Testarossa is a full-fledged and broadly deployed commercial compiler.

10. Conclusion

This paper describes the use of machine-learned models for method-specific compilation in Testarossa, a commercial JiT compiler from IBM. The framework includes (*i*) a lightweight profiling mechanism; (*ii*) a binary archive format for large-scale data collection; (*iii*) tools to process the data collected to train machine-learned models; and (*iv*) a lightweight communication protocol to integrate the compiler and the machine-learning algorithms.

[12] When compared to Jikes, Testarossa has, in fact, two additional optimization levels, but one of those is a special-purpose optimization level (ahead-of-time compilation), not used in this paper.

The experimental evaluation revealed that the learned models outperforms out-of-the-box Testarossa on average for start-up performance, but underperforms Testarossa for throughput performance, one of the key metrics used to measure performance during development, and that thus received significant development attention over many years. A surprising result is the significant reduction in compilation time across two benchmark suites. A pleasantly positive result was the ability of the model to generalize, for start-up performance, from learning in SPECjvm98 benchmarks to DaCapo benchmarks.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at `http://www.ibm.com/legal/copytrade.shtml`.

Java is a trademark of Oracle and/or its affiliates. Intel is a trademark of Intel Corporation or its subsidiaries in the United States and other countries. UNIX is a registered trademark of The Open Group in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

References

[1] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, Effective Code Generation in a Just-in-Time Java Compiler. *SIGPLAN Notices*, 33(5):280–290, 1998.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 47–65, New York, NY, USA, 2000. ACM.

[3] Matthew Arnold, Michael Hind, and Barbara Ryder. An Empirical Study of Selective Optimization. *Languages and Compilers for Parallel Computing*, pages 49–67, 2001.

[4] 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*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] John Cavazos, Grigori Fursin, Felix V. Agakov, Edwin V. Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *Code Generation and Optimization (CGO)*, pages 185–197, San Jose, CA, March 2007.

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

[7] Corinna Cortes and Vladimir Vapnik. Support-vector Networks. *Machine Learning*, 20:273–297, September 1995.

[8] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing frontiers (CF'07)*, pages 131–142, New York, NY, USA, 2007. ACM.

[9] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[10] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. Quick and Practical Run-time Evaluation of Multiple Program Optimizations. *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 34–53, 2007.

[11] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. MILEPOST GCC: Machine Learning Based Research Compiler. In *GCC Developer Summit 2008*, Ottawa, ON, 2008.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

[13] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay 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*, page 12, Berkeley, CA, USA, 2004. USENIX Association.

[14] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Verlag, 2009.

[15] Kenneth Hoste and Lieven Eeckhout. COLE: Compiler optimization level exploration. In *Code Generation and Optimization (CGO)*, pages 165–174. ACM, 2008.

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

[17] IBM Corporation. IBM J9 Java Virtual Machine. `http://www.ibm.com/developerworks/java/jdk/`.

[18] S. Sathiya Keerthi, S. Sundararajan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. A Sequential Dual Method for Large Scale Multi-class Linear SVMs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'08)*, pages 408–416, New York, NY, USA, 2008. ACM.

[19] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online Performance Auditing: Using Hot Optimizations Without Getting Burned. *Programming Language Design and Implementation (PLDI)*, 41(6):239–251, 2006.

[20] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.

[21] R. N. Sanchez. Applying support vector machines to discover method-specific compilation strategies. Master's thesis, University of Alberta, Edmonton, AB, Canada, September 2010.

[22] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using support vector machines to learn how to compile a method. In *22nd International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, Petropolis, RJ, Brazil, October 2010.

[23] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional, 2nd edition, 2005.

[24] Kapil Vaswani, Matthew J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. Microarchitecture Sensitive Empirical Models for Compiler Optimizations. In *Code Generation and Optimization (CGO)*, pages 131–143, Los Alamitos, CA, USA, 2007. IEEE Computer Society.