“All we have to decide is what to do with the time that is given us.”
— Gandalf the Grey

University of Alberta

Applying Support Vector Machines to Discover Just-in-Time Method-Specific Compilation Strategies

by

Ricardo Nabinger Sanchez

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Ricardo Nabinger Sanchez
Fall 2010
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Examining Committee

José Nelson Amaral, Department of Computing Science

Duane Szafron, Department of Computing Science

Richard S. Sutton, Department of Computing Science

Bruce F. Cockburn, Department of Electrical and Computer Engineering

To *Débora*, for her endless love and support.

Abstract

Adaptive Just-in-Time compilers employ multiple techniques to concentrate compilation efforts in the most promising spots of the application, balancing tight compilation budgets with an appropriate level of code quality. Some compiler researchers propose that Just-in-Time compilers should benefit from method-specific compilation strategies. These strategies can be discovered through machine-learning techniques, where a compilation strategy is tailored to a method based on the method's characteristics. This thesis investigates the use of Support Vector Machines in Testarossa, a commercial Just-in-Time compiler employed in the IBM® J9™ Java™ Virtual Machine. This new infrastructure allows Testarossa to explore numerous compilation strategies, generating the data needed for training such models. The infrastructure also integrates Testarossa to learned models that predict which compilation strategy balances code quality and compilation effort, on a per-method basis. The thesis also presents the results of an extensive experimental evaluation of the infrastructure and compares these results with the performance of the original Testarossa.

Acknowledgements

Throughout my research, many people helped me either directly by contributing to research the goals, or indirectly by ensuring favorable conditions. I am thankful for each and every one of you.

Débora, in special, for her constant support and love. My parents, especially my mother *Liane*, for the remote support and for being brave enough to cross the planet and enjoy the Winter in Edmonton. My sister and great friend *Juliana*, for her constant optimism even in the darkest times. My uncle *Bob*, for his support and frequent updates on the weather in Miami.

My supervisor *Nelson* for his mentorship and constant support. Besides being a great supervisor, he is also a great cook. My co-supervisor *Duane* for the extensive advice on machine-learning topics.

The IBM Testarossa JIT Team, especially *Marius Pirvu* and *Mark Stoodley* for their abundant support with Testarossa. The IBM Center for Advanced Studies (CAS) for the funding, which made the research culminating in this thesis possible.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ANN	Artificial Neural-Network
AoT	Ahead-of-Time
API	Application Programming Interface
BCD	Binary-coded Decimal
CPU	Central Processing Unit
FIFO	First-in first-out
IL	Intermediate Language
IPC	Inter-process Communication
JiT	Just-in-Time
JVM	Java Virtual Machine
MMH	Maximum Margin Hyperplane
MSR	Model-specific Register
NN	Nearest Neighbor
PMC	Performance Monitoring Counter
RBF	Radial Basis Function
RL	Reinforcement Learning
RVM	Research Virtual Machine
SDM	Sequential Dual Method
SVM	Support Vector Machines
SVR	Support Vector Regression
TSC	Time Stamp Counter
VM	Virtual Machine
XML	Extensible Markup Language

Chapter 1

Introduction

Dynamic programming languages such as Java^{TM} offer developers a wide range of features, such as the inclusion of new types (Java classes) while the application is running. The success of such languages relies heavily on the underlying runtime support, which in the case of the Java language is provided by the Virtual Machine (VM). Dynamic languages are often associated with lower execution performance, so the VM not only provides the needed runtime support, it also employs many techniques to improve the performance of the application [4, 10].

One of such techniques is Just-in-Time (JiT) compilation, where the application is dynamically compiled (*i.e.*, while the application is running) into native code that can execute directly on the host platform. Modern JiT compilers implement numerous code transformations used in the optimization phase of such compilations in order to further improve the execution performance of the code generated. However, since the compiler competes with the application for the same resources, a balance must be struck in terms of compilation effort versus code quality [1, 6]. JiT compilers include multiple optimization levels (compilation plans) which are selected in reaction to the execution behavior of the application, focusing compilation efforts proportionally to the higher-demanding portions of the application. For example, a few key methods in a Java application can be compiled at the highest optimization level implemented by the JiT compiler, whereas infrequent or short-lived methods may be compiled at the lowest optimization level or not be compiled at all [5].

Designing a compilation plan and selecting the combination of code transformations that should be included requires a significant amount of effort. Currently, each optimization level is hand-tuned by compiler experts during the course of many years across a wide range of platform combinations. With the introduction of new platforms supported by the compiler, the existing compilation plans can require adjustments or even platform-specific versions. Worse, maintaining the optimization levels is difficult because changes in a compilation plan can be beneficial for some applications, but degrade the performance of others [26].

The premise of this study is that compilation plans can be tailored on a per-method basis with the use of machine-learned models. By extracting features describing methods, and presenting the machine-learning algorithm with multiple variations of the original compilation plan, it should be possible to create a model that identifies patterns in the data presented and predicts a method-specific compilation plan. The learned model can take into consideration, for example:

- When a code transformation is harmful (leading to slower executing code) and should be disabled for specific methods;
- When code transformations can be disabled in the compilation plan, while delivering equivalent code performance, at a reduced compilation cost;
- When the original, hand-tuned compilation plan is the best decision for the method.

This thesis presents a complete framework that integrates a method-specific decision mechanism based on machine-learned models in Testarossa, an enterprise-grade, commercial Just-in-Time compiler for Java from IBM® used in the IBM J9^{TM} Java Virtual Machine. Specifically, the main contributions of this thesis are:

- A data collection infrastructure, that performs compilation experiments using a light-weight method profiling mechanism.
- A customized binary archive format to facilitate large-scale data collection experiments.
- Development of supporting tools to convert archives into the format required by the machine-learned models used in this study.
- A lean communication protocol based on Inter-process Communication (IPC) to integrate the machine-learned models with the compiler. The communication approach allows different machine-learned models to be easily swapped without changes to the compiler.
- Evidence that, for some applications, method-specific compilation produces code with performance comparable to the compilation plans currently in use, however, using less compilation time.

Just-in-Time compilers, including IBM Testarossa, are discussed in Chapter 2. Machine-learning models are introduced in Chapter 3. Chapter 4 presents the data collection infrastructure in great depth. The techniques used to explore different compilation plans are discussed in Chapter 5. The steps needed to train a machine-learned model are detailed in Chapter 6, followed by the approach used to integrate a learned model with the compiler, in Chapter 7. Chapter 8 reports on the experimental results obtained with the proposed solution. Similar approaches are discussed in Chapter 9. Chapter 10 presents concluding remarks. Finally, Chapter 11 describes future work.

Chapter 2

IBM Testarossa

Since its introduction in 1995, the Java programming language has been used in a wide range of applications. Part of its popularity can be attributed to the use of a Virtual Machine (VM) that allows for greater application portability. As long as a VM is available, the application can be executed regardless of the underlying architecture.

Java has also been the focus of extensive research in order to deliver improved execution performance, especially when it comes to the VM implementation and additional techniques that can accelerate the execution speed of applications. JiT compilation is a technique where the Java bytecodes are translated to another instruction set that can execute natively instead of being interpreted. The JiT compiler exploits features in the underlying architecture and, often, delivers improved execution performance.

In the course of this translation, many code-transformation opportunities arise to further improve the execution performance of a Java application. Thus, a JiT compiler acts as a full-fledged compiler. A JiT compiler competes with the application for resources. Therefore, the conscious use of the resources during compilation is of major importance. Otherwise, all benefits from advanced JiT compilation could be outweighed by the costs of the compilation.

This chapter introduces the JiT compilation technique in Section 2.1, followed by a discussion, in Section 2.2, on the compiler used in this research, Testarossa, from IBM.

2.1 Just-in-Time Compilation

A Java application is generated by compiling the source code into Java bytecodes. This bytecode representation targets a hypothetical architecture, the Java Virtual Machine (JVM) [17]. This virtual machine implements a stack-based execution model, where all operations (instructions of the hypothetical architecture) are performed on a stack that represents the memory. Because Java applications are compiled for a JVM, they are considered *portable*. The JVM is a native application that executes Java bytecodes by interpreting this code on the host platform.

Most modern JVMs implement a dual model: (*i*) a regular bytecode interpreter, and (*ii*) an acceleration mechanism based on JiT compilation. The interpretation follows the semantics defined for the JVM and performs the equivalent bytecode operations that are intended for a stack-based architecture onto the host platform, which may use a different architectural model.[1] The need for a JiT compilation mechanism arises from the desire to improve execution performance—especially on portions of the application that are very computation-intensive (*e.g.*: cryptography, encoding/decoding of sound, image, and video). The performance is improved by compiling Java bytecodes into native instructions for direct execution on the host platform, thus eliminating the bytecode interpretation overhead.

JiT compilers can be broken down 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 JVM uses this profiling information to decide if it is worthwhile to compile a method to native code, while the JiT compiler also uses the profiling information to decide how to optimize the method.

2.2 The IBM Testarossa Just-in-Time Compiler

Testarossa is a state-of-the-art JiT compiler employed in the IBM J9$^{\text{TM}}$Java Virtual Machine, implementing numerous optimization techniques [18, 22]. Because the compilation comes at a significant cost in terms of overhead, Testarossa implements multiple mechanisms to identify portions of the Java application that can benefit from JiT compilation. The goal is to avoid outweighing the benefits of JiT compilation due to the compilation overhead.

Testarossa implements adaptive multi-level compilation of individual methods in the application, reacting to profiling information dynamically generated from the execution of the application. Methods can be recompiled multiple times after they are compiled for the first time. Testarossa implements five optimization levels identified by adjectives related to temperature: *cold* (lowest optimization level), *warm*, *hot*, *very hot*, and *scorching hot* (highest optimization level). The temperature refers to the estimate of how frequently a method is executed.[2] The hotter a method is, the more compilation effort Testarossa invests in it in the hopes of generating faster code.

Testarossa uses a combination of invocation counters and time sampling to estimate the hotness of a method. This combination of counters and samplings enables Testarossa

[1]Most contemporary computers implement a load-store register-based model, where all computations are performed on architecture registers and the memory is the storage. The computation requires load-store cycles, where data is loaded from memory into registers, and then stored back to memory after the computation is complete.

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

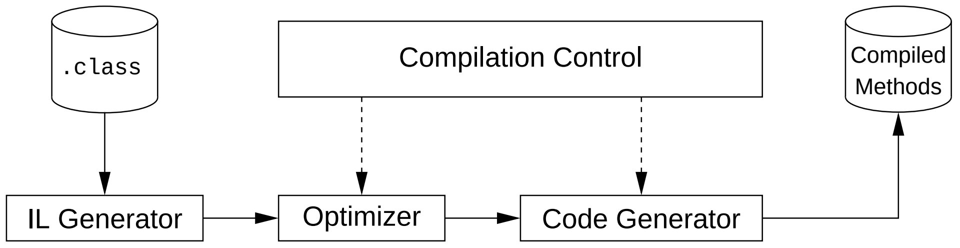

Figure 2.1: The four major components in IBM Testarossa. The Intermediate Language (IL) Generator translates a method from Java bytecodes into the intermediate representation used by the Optimizer during the compilation. After an optimization phase, the Code Generator produces a native version of the method that can execute on the host platform instead of being interpreted. The Compilation Control, in turn, decides when to compile or recompile a method and at which optimization level.

to anticipate the compilation of methods that spend a significant amount of time in fewer invocations. The estimation mechanism also allows for recompilations at a higher optimization level if a method executes frequently with regard to the pool of methods active in a Java application.

An overview of the architecture of Testarossa is presented in Figure 2.1. There are four major components, represented as boxes in the figure. The **Intermediate Language (IL) generator** converts Java bytecodes loaded from Java class files into a tree-form intermediate-language representation. This representation is used as both input and output by the optimizer. 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, s390, 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 the profiling information gathered during the execution of the application. Most of the resources necessary for compilations are spent in the optimization stage.

The optimization levels implemented by Testarossa are organized as ordered sets of code transformations (a *compilation plan*) that are applied on the IL-tree of the method being compiled. Each transformation can be subject to compilation flags that the optimizer tests before applying them. These flags consist of characteristics of the method being compiled (*e.g.*: whether the method has loops) or if a code transformation is disabled (*e.g.*: user request). With this mechanism, Testarossa can save compilation time and resources by slightly adjusting the compilation plan for the method.

The smallest compilation plan in Testarossa, for the *cold* optimization level, includes over 20 transformation passes. In contrast, the largest compilation plan (for the *scorching hot* optimization level) has more than 170 transformation passes. These figures include code transformations that are applied more than once in the compilation plan. For example, deadTreesElimination and treeSimplification are applied multiple times as cleanup steps.

In Testarossa, compilations are performed by at least one dedicated thread, running in the background. Normally, the compilations are asynchronous to the application execution. In some cases, however, the application must have its execution halted until the compilation of a method completes. For example, if an application invokes a method that is part of a class hierarchy which just changed in a non-compatible way (*e.g.*: a class dynamically loaded by the application overrides methods from the parent classes), the method must first complete compilation before the JVM is allowed to continue execution.

In this thesis, the compilation plans included in Testarossa are subject to explorations with the goal of fine-tuning them on a method-specific basis. This is done in conjunction with a machine-learned model, Support Vector Machines, discussed in Chapter 3.

Chapter 3

Support Vector Machines

Support Vector Machines (SVM) are statistical learning models that work by finding maximum separating hyperplanes[1] in a training data set composed of data instances and their respective values [11, 19]. A data instance is a p-dimensional feature vector $\vec{x}$ representing an i-th observation (*e.g.*: code size and presence of loops in a method to be compiled). The value of a data instance is either a label[2] (*e.g.*: a code transformation was applied to the method) or an output (*e.g.*: measured speedup of a code transformation).

Support Vector Machiness (SVMs) are trained for either regression or classification. Regression SVMs[3] are used to estimate the output for an unseen feature vector $\vec{X}$ by measuring the distance from $\vec{X}$ to the separating hyperplane. Regression SVMs are not discussed further because they are not used in this study. A classifying SVM predicts the class of $\vec{X}$ by computing the location of $\vec{X}$ relative to separating hyperplanes and by inspecting the sign of the classification function obtained from this computation.

Section 3.1 discusses SVMs used for classification problems.

3.1 Classifying Support Vector Machines

The simplest classifying SVM deals with a single class, and with a training data set consisting of N instance pairs $(\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots, (\vec{x}_N, y_N)$, where each $\vec{x}_i$ is a p-dimensional vector $\vec{x}_i \in \Re^p$ and, $y_i \in \{-1, 1\}$. Training a classifying SVM consists of finding a hyperplane that maximizes the margin M, as illustrated in Figure 3.1. In the case of linearly separable data, such as shown in Figure 3.1(a), the margin M is the distance between the separating hyperplane (solid line) and a point in either class (hollow or filled circles).

The closest instances to the separating hyperplane are the **support vectors** (dashed lines). When the data is not linearly separable, as illustrated in Figure 3.1 (b), some amount of misclassification is inevitable in order to properly place the separating hyperplane. SVMs

[1]The term Maximum Margin Hyperplane (MMH) is also common in the literature.

[2]Labels are often called *classes*.

[3]Regression SVMs are also referred to as Support Vector Regression (SVR).

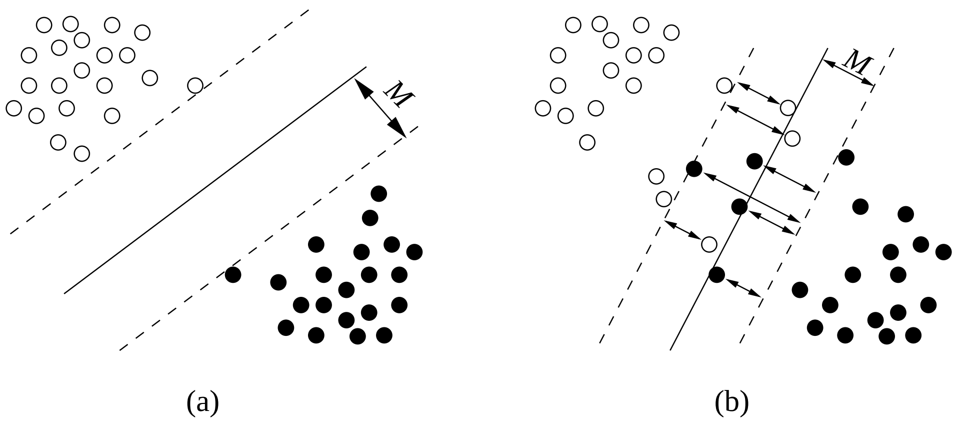

Figure 3.1: Example of hyperplane placement. When the data is linearly separable (*a*), the separating hyperplane (solid line) is placed as far as possible from either class (hollow and filled circles) with a margin of M, and the support vectors (dashed lines) lie at the instances closest to the hyperplane, from both classes. When the data is overlapping (*b*), some amount of misclassification must be allowed (arrows) during hyperplane placement.

can tolerate this condition with **soft-margins**, by maximizing the margin M at the same time that the misclassification error is minimized.

A separating hyperplane is defined as a p-dimensional vector $\vec{X}$ that satisfies

$$\vec{W} \cdot \vec{X} + b = 0, \tag{3.1}$$

where $\vec{W}$ is a p-dimensional vector of weights, b is an additional weight (bias), and $\cdot$ denotes a dot product operation. If the data is linearly separable, then all elements of the training set satisfy the following relations:

$$\begin{aligned} \vec{W} \cdot x_i + b &\geq 1 \qquad \text{if } y_i = 1, \\ \vec{W} \cdot x_i + b &\leq -1 \quad \text{if } y_i = -1, \end{aligned} \tag{3.2}$$

for a vector $\vec{W}$ and a scalar b [11]. When $\vec{W} \cdot x_i + b = 1$, the data instance x_i lies at the support vector. The direction of maximal separation between the two classes is given by $\frac{\vec{W}}{\|\vec{W}\|}$.

The classifying SVM is expressed as the following quadratic program

$$\begin{aligned} \underset{W,b}{\text{minimize}} \quad & \frac{1}{2}\|\vec{W}\|^2 + C\sum_{i=1}^{N}\xi_i \\ \text{subject to} \quad & \vec{W} \cdot \vec{x_i} + b \geq 1 - \xi_i, \quad i = 1, \ldots, p \\ & \xi_i \geq 0, \qquad\qquad\qquad i = 1, \ldots, p \end{aligned} \tag{3.3}$$

The maximum margin between the classes is expressed by $\frac{1}{2}\|\vec{W}\|^2$. To deal with overlapping data, the classifying SVM includes the sum of deviations $\sum_{i=1}^{N}\xi_i$. The parameter C controls the amount of penalization given to deviations (the cost of accepting a misclassification in the model). A small value (*e.g.*: 0.1 or less) allows more misclassifications, whereas large values (*e.g.*: 10 or more) amplify the misclassification cost. Values too small produce poor models, and values too large can cause the model to overfit. The value of C cannot be computed beforehand, thus it is set to a value suggested by an expert or experimentally determined by evaluating several classifiers using different values of C.

Multi-class classification SVMs, such as those used in this study, are trained in two different ways: (*i*) one-versus-one and (*ii*) one-versus-all. In a one-versus-one approach, classifiers are trained for each pair of classes and each classifier casts a vote whether the data point $\vec{X}$ belongs to that class or not. The class with the majority of votes is selected. 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 $\vec{X}$ to the given classifier boundary.

SVMs can be further extended and adapted to different scenarios with the use of kernel functions, which are detailed in Section 3.2.

3.2 Kernel Functions

In many applications of SVMs the data is not linearly separable in the original feature space. In those cases, an SVM can be built using a **kernel function** that maps the original feature space into a higher-dimensional space, where the data becomes linearly separable.

A kernel function K is a symmetric function with the form $\phi : \Re^p \rightarrow \Re^P$. K maps the p-dimensional space into a P-dimensional space given by

$$K(\vec{X}_i, \vec{X}_j) = \phi(\vec{X}_i) \cdot \phi(\vec{X}_j), \tag{3.4}$$

where $\vec{X}_i$ and $\vec{X}_j$ are p-dimensional vectors. The classification function G is updated accordingly:

$$G(\vec{X}) = w \cdot \phi(\vec{X}) + b, \tag{3.5}$$

where the data point $\vec{X}$ is transformed by ϕ prior to the application of the classification function [11]. Table 3.1 presents kernel functions commonly found in the literature.

When the original p-dimensional feature space is transformed into the P-dimensional space and the separating hyperplanes are computed, often the hyperplanes turn out to be non-linear when mapped back to the original space. This characteristic gives SVM a high generalization power, including cases where the data is organized in arbitrary shapes. Figure 3.2 illustrates a possible scenario. The original p-dimensional feature space on the left is transformed into a P-dimensional space by a kernel function $K(\vec{X}_i, \vec{X}_j)$. In the transformed space, the data is linearly separable by a hyperplane. The hyperplane (solid line) and the support vectors (dashed lines) have a non-linear shape when mapped back into the original space.

Table 3.1: Common SVM kernels found in the literature.

Kernel function	**Definition**
d-degree polynomial	$K(\vec{X}_i, \vec{X}_j) = (\vec{X}_i \cdot \vec{X}_j + 1)^d$
Gaussian radial basis function	$\exp\left(-\frac{\lVert \vec{X}_i - \vec{X}_j \rVert^2}{2\sigma^2}\right)$
Sigmoid	$\tanh(\kappa_1 \vec{X}_i \cdot \vec{X}_j + \kappa_2)$

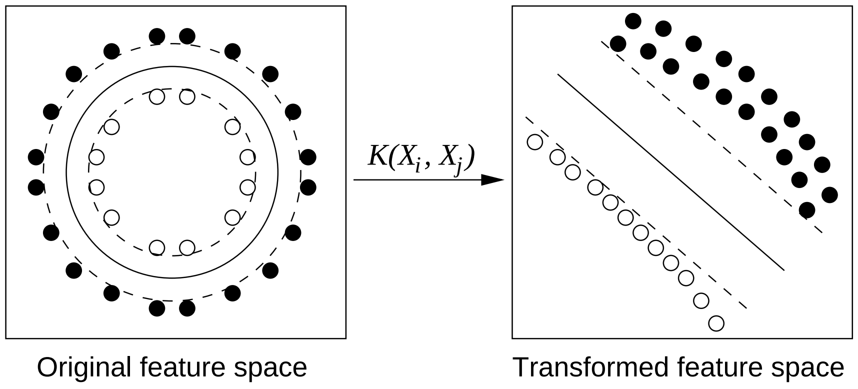

Figure 3.2: Example of feature space transformation produced by kernel functions. The original feature space on the left is transformed with a kernel function $K(\vec{X}_i, \vec{X}_j)$, where the data is linearly separable by a hyperplane, but is non-linear when mapped back to the original space. The solid line represents the hyperplane, and the dashed lines the support vectors.

The advantages and limitations of using SVMs for classification problems are discussed in Section 3.3

3.3 Advantages and Limitations of SVMs

The key advantage of SVMs is their flexibility to adapt to many different situations. The soft-margins allow SVMs to deal with 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. The 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. Lastly, the possibility of using kernel functions, when the training data is not linearly separable in the original feature space, makes SVMs very versatile, allowing even the use of custom kernels.

SVMs do have drawbacks, however. The selection of an appropriate value for the C parameter requires multiple training iterations and may increase training times significantly. Choosing a kernel function (if any) can also be difficult. Often the training data is too complex to estimate which kernel function is appropriate. In such cases, additional training experiments must be performed, further increasing training times. Moreover, if a testing data set is not available the best performing kernel function can only be determined after deploying the learned model.

The kernel function can be responsible for a large amount of computation overhead, which can hinder its use with complex or high-dimensional training data. In addition, if the original feature space is already high-dimensional then mapping the data into a higher-dimensional space may not improve the quality of the learned model: it only results in unnecessary computation. When the feature space is high-dimensional, an *identity* kernel function may be successful.

The SVM implementation used in this study is discussed in Section 3.4.

3.4 SVM Implementation

The SVM implementation used in this study, LIBLINEAR [13], supports the training of several SVM classifiers, including a multi-class SVM. The multi-class classifier implementation in LIBLINEAR uses a variant of the Sequential Dual Method (SDM) [24], which performs better on large-scale classification problems than the one-versus-all approach. Large-scale problems include those that have numerous instances and/or distinct classes, such as the one presented in this study.

The resulting model learned by LIBLINEAR consists of a $p \times L$ matrix. The matrix contains the contributions of each of the p dimensions of the feature space for separating the L distinct classes present in the training data set. The prediction time is proportional to the size of the matrix.

Both trainer and classifier are implemented in a mixture of C and C++ languages. These implementations share components with another SVM implementation from the same authors, called LIBSVM. Like LIBLINEAR, LIBSVM implements multiple classification methods, including the multi-class classifier. The main difference is that LIBSVM focuses on a wider selection of kernel functions, with many non-linear kernels, including those listed earlier in Table 3.1.

In order to learn a multi-class SVM classifier, a large-scale data-collection process must be performed. Chapter 4 describes the infrastructure implemented for collecting data that is then used to generate a training data set.

Chapter 4

Data Collection

Data collection is an essential step in generating data that can be used to train a machine-learned model. The collection process mimics the regular operation of Testarossa, while performing controlled changes in the compilation plans to explore alternative scenarios on a per-method basis.

This chapter discusses the workflow when executing Testarossa in data-collection mode, starting with an overview in Section 4.1. Section 4.2 describes the features that characterize a method and how these features are collected. Next, Section 4.3 presents how compilation-plan modifiers are selected and managed during the data-collection process. The profiling mechanism is described in Section 4.4. The storage approach when data collection is complete is detailed in Section 4.5. Section 4.6 discusses alternatives to the mechanisms used, especially profiling and storage, and Section 4.7 concludes the Chapter.

4.1 Architecture Overview

The architecture of the framework for data collection is summarized in Figure 4.1. The individual stages of the data-collection process are detailed in the following sections.

Data collection starts when Testarossa selects a method for compilation (Figure 4.1 (a)). When Testarossa invokes the optimizer for a given compilation, the method features are computed immediately before the optimization starts (b). Then, a compilation-plan modifier is retrieved (c) and combined with the original compilation plan, instructing the optimizer 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 amongst them (which can either cause compiler malfunction or generation of invalid code). Thus, transformations are only removed from the original compilation plan. The method is instrumented in addition to the optimization, enabling it to collect dynamic information, such as execution time, at each invocation.

When the compilation is complete, the freshly compiled method is inserted in the pool

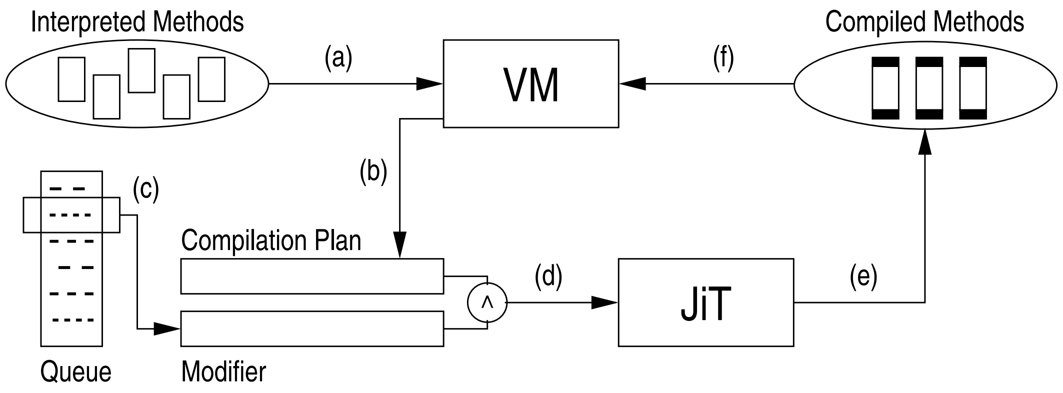

Figure 4.1: Data Collection during training.

of compiled methods (e). It is possible that Testarossa is recompiling a method at a higher optimization level (f). In such cases the resulting method replaces the old version of the method in the pool of compiled methods. In addition, the instrumentation mechanism triggers recompilation requests to the VM, enabling the data-collection process to explore a large number of compilation-plan modifiers in a single JVM execution.

Section 4.2 discusses how method features are computed.

4.2 Computing Method Features

Method features characterize a method by summarizing it in the form of a feature vector $\vec{F}$. Ideally, $\vec{F}$ should contain enough information about a method to allow a machine-learned model to correlate this information to a specific compilation plan that yields higher performance when the method is compiled.

The information contained in $\vec{F}$ is dynamically extracted from the compiler, just prior to the optimization stage. This way, the set of features extracted is consistent in both data-collection and production modes. Not every piece of information made available by the compiler is useful for the training of the model. Compiler developers are in the most-favorable position to indicate what portions of the data can be useful to discover better compilation plans.

In this implementation of method-specific learning, the feature vector $\vec{F}$ is composed of 71 numerical attributes (dimensions). This data can be organized in two sets:

1. **Scalar features** consist of counters and binary attributes for a given method without any special relationship;

2. **Distributions** characterize the actual code of the method by discriminating across operand types and by aggregating similar operations.

Table 4.1: Set of scalar features collected. The features collected are grouped into counters and attributes, where attributes are simple binary features.

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

The two sets of features collected also differ in the way that they are computed, and they are detailed in Sections 4.2.1 and 4.2.2, respectively.

4.2.1 Scalar Features

Scalar features are a subset of the information normally generated by Testarossa on every method compilation. The information carried is very diverse, ranging from counters (*e.g.*: number of parameters received by the method when invoked) to binary attributes (*e.g.*: whether the method is a constructor).

Table 4.1 illustrates the scalar features that are part of the feature vector $\vec{F}$. In the table, scalar features are organized as two logical groups: counters and attributes. The first counter type (*Exception handlers*) indicates how many exception handlers are present in the method. The next two counters, *Arguments* and *Temporaries*, decompose the total number of symbols referenced in the method, respectively, into the number of arguments received by the method when it is invoked, and the number of temporaries used in the method (*e.g.*: local variables). In Testarossa, the number of temporaries t is given by $t = |\{S\}| - |\{A\}|$, where $|\{S\}|$ is the size of the set of symbols referenced in the method, and $|\{A\}|$ is the size of the set of arguments. The last counter, *Tree nodes*, provides the number of nodes in the initial tree that represent the method internally in the intermediate language used by Testarossa.

The second logical group of scalar features are *attributes*. All attributes are binary (represented with a ? next to the name of the attribute in Table 4.1). The attributes are separated into subgroups. In the first subgroup are attributes that are usually made explicit when the method is implemented, either by the semantics of the programming language (*constructor*) or by keywords (*final*, *protected*, *public*, *static*, *synchronized*).

The second subgroup contains three attributes related to the presence of loops. After the method is compiled, loop constructs from the programming language are lost, and the presence of loops is detected by inspecting the instruction stream for backward branches. When a method has a backward branch in the instruction stream, it is marked as possibly having a loop (*may have loops* attribute). Testarossa also employs thresholds to estimate whether the method has loops with many iterations (*many-iteration loops* and *may have many-iteration loops* attributes), which is sometimes the case with nested loops. Loop attributes enable Testarossa to apply loop transformations to a method only if it is eligible (*i.e.*, the method does have loops), saving compilation time otherwise.

The next subgroup contains a diversified set of characteristics about the method. For example, Testarossa has specific optimizations targeting methods that allocate memory dynamically, which can be determined using *escape analysis*.[1] The *unsafe symbols* attribute is set on methods that inline a method from the class `sun.misc.Unsafe`, restricting the applicability of optimizations such as redundant-load elimination.[2] Testarossa detects several methods and classes that perform computations with arbitrary precision, using the core Java library `java.math.BigDecimal`, reflected in the *Uses BigDecimal* attribute. Computations using `BigDecimal` may not be eligible for rematerialization because the code generated can outweigh the benefits of this optimization.[3] The *Virtual method overridden* attribute refers to the condition where the class hierarchy of the method changed in ways that Testarossa could not anticipate. For instance, Java allows classes to be dynamically loaded, and these classes may contain methods that override parent methods. If the change in the class hierarchy invalidates a compiled method, the method is recompiled taking the new class hierarchy into consideration.

The final subgroup records whether the method performs any floating-point computation (*Uses floating point* attribute) and if the JVM should enforce strict compliance of floating-point computations.

In conjunction with the scalar features described in this Section, distributions complete the set of data that comprise the feature vector $\vec{F}$, and are discussed next in Section 4.2.2.

4.2.2 Distributions

Distributions characterize different aspects of the abstract representation of the method in Testarossa. There are two distinct sets of distributions collected: (*i*) distribution over types and (*ii*) distribution over operations. This separation allows for (*a*) a simpler implementation for the collection process, (*b*) a reduced set of features, and (*c*) a smaller storage

[1] *Escape analysis* is used by the compiler to determine the lifetime of an object. For example, if an object can be proven to not escape a given context (*e.g.*: the lifetime of the object is restricted to a method), the compiler can allocate it on the stack, thereby avoiding the heap.

[2] *Redundant-load elimination* is a code transformation that removes a load instruction if it can be proven that the operand is available on one of the architectural registers. In some cases, the load instruction is replaced by a register-to-register copy.

[3] *Rematerialization* is a code transformation where the compiler emits code that recomputes a value when the benefits are evident for the compiler. For example, rematerialization can be used to reduce register pressure.

Table 4.2: Types characterized during distribution data collection. Most types are native to the Java language. Testarossa has internal support for specialized types in addition to the native types.

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`			

requirement.

Each distribution set has a different counter capacity. "Distribution over types" are 16-bit counters (ranging from 0 to $65,535$), while "distribution over operations" are 8-bit (0 to 255). The counters used in the distribution over types are larger because a single operation can increment the counter more than once, depending on the number of operands. Once a counter reaches its maximum possible value, it is no longer incremented.

Table 4.2 presents all of the 14 types that are counted during data collection. Most types characterized in this distribution are native types of the Java language, organized as scalars and non-scalars. Scalars are primitive types that can be either integers (`byte`, `char`, `short`, `int`, and `long`), floating-point (`float`, and `double`), or no type as far as the programming language design is concerned (`void`). The non-scalars native types can be either an address (an array with one or more dimensions) or an object.[17]

Testarossa has explicit support for specialized types in addition to native types. "Long double" is a quadruple-precision 128-bit IEEE-754 floating point type. Both packed and zoned decimals provide support for Binary-coded Decimal (BCD), which are frequently used in financial applications. These types are especially useful when dealing with currencies, where any errors due to data representation would be a liability. In addition, BCD enables efficient fixed-point, arbitrary-precision computation.

Table 4.3 presents the 38 operations that are characterized when computing distributions during data collection of a method. The operations can be grouped into several logical groups, the first one being arithmetic and logic operations (column ALU in the table). Most operations in the ALU group are requirements of any JVM implementation, but Testarossa has support for extra operations when the method is converted into the intermediate representation, especially `compare`. Internally, `compare` is specialized for all types supported by Testarossa and the common predicates (equal, not equal, less than, greater than or equal, greater than, less than or equal).

The second column in Table 4.3, *Cast*, lists the typecast operations that are characterized. As with ALU operations, most types are native to the Java language and those

Table 4.3: Operations characterized during distribution data collection. Most operations are native for the instruction set of the Java language, but Testarossa has internal support for some specialized operations in addition to the native ones.

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`*		
	`check`		***Mixed operations***

typecasts are requirements for any JVM. However, as Testarossa supports additional types (*e.g.*: `longdouble`, `packed` and `zoned` decimals), it has additional typecast operations that are characterized during data collection. The typecast operation makes explicit the intended type after the conversion (*i.e.*, the `int` typecast operation converts any valid operand into `int`).

Next in the table are load and store operations. The instruction set for the Java language contains many different opcodes for loading and storing data [17]. In particular, every primitive type has a general instruction for load, load constant, and store operations, and an additional set of specialized instructions. The general form of an instruction specifies the operation and requires an operand indicating the index of the local variable affected by the operation, whereas the specialized forms embed the operand index in the opcode. For example, the `iload`[4] instruction requires an operand while `iload_1`[5] does not. Testarossa expresses these operations in the general form when converting a method to its internal representation, but in a specialized form for all types supported directly by Testarossa. During data collection, the specialized operations are coalesced into either load, load const, or store operations because, in practice, only a small subset of type-specialized instructions are used in a method. The distinction across the operations is reflected in the distribution of types for the method: each type-specialized form triggers a different type counter.

The group *Memory* in Table 4.3 refers to the characterization of distinct dynamic memory-allocation operations. The key difference amongst all three operations is the purpose of each one: (*i*) `new` is used to create an instance of a single object; (*ii*) `new array`

[4]An `iload` is an integer (`int`) load from variable index specified as operand to the top of the stack.
[5]An `iload_1` is an integer (`int`) load from variable index #1 to the top of the stack.

allocates space for a one-dimensional array of elements of a certain type; and (*iii*) `new multiarray` is used to allocate a multi-dimensional array of elements of a certain type.

The last column in the table starts with the *JVM* group. The first counter is incremented when `instanceof` Java instructions are found, which are used to test whether an object reference in the stack is of a given type. The second counter, `synchronization`, counts the number of serializing Java instructions (`monitorenter` and `monitorexit` instructions) in the method. Lastly, the number of exceptions explicitly thrown (by using the Java instruction `athrow`) is reflected in the counter `throw`.

Next, the *Branch* group contains counters for branching operations. Testarossa supports a wide range of fine-grained branching operations, but in practice only a handful is used in the scope of a single method. Therefore, the majority of the branching operations are coalesced in a single `branch` counter. On the other hand, method invocations are handled in a separate counter, `call`. The intermediate representation in Testarossa makes the return type of the method being invoked explicit, triggering the appropriate type counter during data collection.

Last in Table 4.3 are two groups that are counters on their own. The *Array operations* counter is incremented whenever an array-specific operation is performed at the intermediate representation level in Testarossa. Array operations include bounds check, array copy, and array comparison. They do not include the modification of an element in an array. Such operation involves loading an element from the array (thus a *Load/Store* operation), performing a computation (an *ALU* operation), and then storing the result back (another *Load/Store* operation).

The approach used for computing the distribution counters is detailed in Section 4.2.3.

4.2.3 Computing Distributions

The 52 distribution counters, split in two groups (types and operations), are computed in a single pass over the tree-based representation of the method in Testarossa, just prior to the start of the optimization stage. The implemented function traverses this tree and increments the appropriate counters by using a lookup table (comprising every possible intermediate representation node in Testarossa) to decode each node. Nodes that are not relevant for the computation of the distributions are silently ignored during the traversal.

Algorithm 4.1 presents the pseudo-code for the implementation in use for computing distributions over both types and operations. Every node of the input tree is touched only once on a single linear scan (line 1). Each node has its opcode decoded into a pair $\langle T, O\rangle$, where T is the index of the appropriate counter in the types distribution set and O the index in the operations distribution set (line 2). The decoding function is implemented as a lookup table, where every intermediate level opcode in Testarossa is mapped to the corre-

Algorithm 4.1 Single-scan computation of distributions over types and operations.

```
Require: Intermediate level representation of the method $tree$
1: for all $node \in tree$ do
2:     $\langle T, O \rangle \leftarrow decode(node.opcode)$
3:     if $counting(T) \wedge types[T] < 65,535$ then
4:         $types[T] \leftarrow types[T] + 1$
5:     if $counting(O) \wedge operations[O] < 255$ then
6:         $operations[O] \leftarrow operations[O] + 1$
```

sponding pair of indexes for each set of distributions. Some of these opcodes refer to internal Testarossa mechanisms (*e.g.*: nodes that mark where a specific instrumentation hook is to be inserted by the code generator), and thus are not of interest for learning a model. In these cases, the respective entry in the lookup table is set to an invalid index, signaling that the opcode should not be counted. During the computation of the distributions, both indexes are tested to confirm that they should be counted and, if so, whether the respective counters will not overflow (lines 3 and 5). If the conditions are met, the respective counters are incremented by one (lines 4 and 6). Otherwise, the counters remain at their maximum capacity, preventing overflows.

The distributions are computed last during feature collection for the method being compiled. The next step in the data collection, which is explained in Section 4.3, is to select a compilation-plan modifier to be used by the compiler.

4.3 Selecting a Compilation-plan Modifier

After the features for the method were successfully collected, and before the optimizer can be allowed to start, a compilation-plan modifier must be selected in order to perform a compilation experiment on the method. Two key factors are taken into consideration when choosing a compilation plan:

1. Whether a special compilation-plan modifier must be used;
2. Expiring compilation-plan modifiers after they have been used for a sufficient number of times.

Both factors are detailed in the following sections.

4.3.1 Special Compilation-plan Modifier

One consideration when selecting compilation plans is to include a *null modifier*, which is a special compilation-plan modifier that does not change the original compilation plan in Testarossa in any way. The hypothesis is that the original plan in Testarossa can be, indeed, the ideal compilation plan for a method being compiled. In practice, the hypothesis is confirmed and a machine-learned model frequently chooses the *null modifier* for many computation-intensive methods.

During data collection, the experiment on which the *null modifier* is used can be defined arbitrarily, depending on the method used to generate the compilation-plan modifiers. Therefore, the *null modifier* can be the very first experiment performed, or it may be applied late in the experimentation process. There are advantages and disadvantages in both cases.

On the one hand, issuing the *null modifier* ensures that the original compilation plan in Testarossa is experimented over all methods. If it turns out to be the most advantageous plan for a large number of methods, it is very likely that the machine-learned model will recommend it frequently. On the other hand, issuing the *null modifier* at an arbitrary later time can be useful for methods that are very infrequent (*i.e.*, a method that is only occasionally compiled in a few data collection experiments). In such cases, there is a chance that the full-fledged compilation plan will not compensate the effort invested by optimizing the method if it is invoked only a few extra times. If this is also true during data collection, the method may be compiled with different compilation plans, which could provide a better tradeoff in terms of reduced compilation effort with equivalent execution performance.

The placement of the *null-modifier* plan during data collection is an exploration setting, which depends on the objectives of the exploration. The exploration approaches are explained in Chapter 5. For the moment, it suffices to know that the special plan is used either as the very first plan modifier, to ensure that it is used on all methods, or as the third one in more aggressive searches, to ensure that other modifiers are always tested but with a good chance of the *null modifier* being used as well.

As the exploratory compilation is carried out, it is important to retire compilation-plan modifiers so that the search for alternate compilation plans can make progress. Section 4.3.2 details the expiration mechanism of compilation-plan modifiers.

4.3.2 Expiring Compilation-plan Modifiers

The second key factor when selecting a compilation-plan modifier is to expire modifiers that have been used in a number of compilations. The number of times a modifier is used is a parameter for the exploration. The expiration of modifiers ensures that the exploration progresses, since each unique method has a full space of compilation-plan modifiers to explore.

When in data-collection mode, the descriptor of a method in Testarossa is augmented to store the index of the last compilation-plan modifier used. Testarossa is also extended with a control component, called *strategy control*, that is responsible for preparing all compilation-plan modifiers to be explored and for controlling their expiration.

Each time a method compilation starts, a compilation-plan modifier is requested by means of a token. This token specifies the index of the modifier to be used, or 0 if this is a first-time compilation for the method. The *strategy control* verifies that the token is still valid before responding with the modifier. If that modifier is not expired, it is used for the compilation of the method, and when the compilation is complete the number of times that this modifier was used is updated. Moreover, the *strategy control* issues the next token that the method should use on a future recompilation, which is the next compilation-plan modifier that has not expired yet.

When a compilation requests a modifier using an expired token, the compilation is automatically assigned a different, unexpired, token and the respective modifier. The expiration rate of modifiers is a parameter of the exploration approach. In some cases modifiers should not expire, which is achieved with an expiration rate of zero. For instance, exploration approaches focusing on fine-tuning the original compilation plans deployed in Testarossa do not benefit from the expiration of modifiers.

With the compilation-plan modifier selected, the optimizer is allowed to start and the compilation to progress. Before the code generation, the method is modified by means of *instrumentation*, as detailed in Section 4.4.

4.4 Instrumentation of Methods

When Testarossa is operating in data-collection mode, the internal data structure that represents a compiled method is augmented with fields used by the instrumentation. These fields collect dynamic information such as the time spent on each invocation of the method. Throughout this document, these additional fields are referred to as the *instrumentation data block*. Methods are instrumented before code generation by modifying the intermediate-level tree that represents the method throughout the compilation in three ways:

1. The first basic block of the method (its entry point) is modified by inserting a call to the initial time collection function (`TR_jitPTTMethodEnter`). This call becomes the very first operation in the method. This is called the *enter hook* and it is responsible for taking an initial timestamp and storing it in the instrumentation data block.

2. Each basic block containing a return operator is also modified by inserting a call to the exit time collection function (`TR_jitPTTMethodExit`). This call becomes the last operation before the method's return. This is called an *exit hook*, responsible for computing the time spent in a given invocation of the method, and also to increment the invocation counter by one. The difference between the current timestamp and the one obtained by the entry hook is stored in the instrumentation data block.

3. Basic blocks that *throw* an exception are instrumented with a call to `TR_jitPTTMethodExit`. The time spent in the method is computed and stored in the instrumentation data block, and the invocation counter incremented.

Two time measurements are recorded during data collection: (*i*) the compilation time and (*ii*) the running time of a compiled method. The compilation time is already available in Testarossa. Thus it is simply polled after the compilation is complete and stored.

In order to measure the time spent in a method, we found it to be necessary to use an approach tailored for Intel x86 architectures on both Linux and Windows platforms. The main reasons for this customized approach is to avoid the overhead imposed by the measurement itself, and to be able to take high-resolution timestamps. The overhead must be kept as low as possible because data collection usually happens over all invocations of all compiled methods in the application. A high-resolution mechanism is necessary to enable the measurement of methods with very short execution time. Methods that perform a very small set of operations (*e.g.*: getter/setter methods) are fairly common in programs written in object-oriented languages. Therefore a high-resolution measuring mechanism is required to actually capture such events.[6]

On x86 architectures, the Time Stamp Counter (TSC) was featured in the early Intel Pentium, becoming widely available shortly after its introduction. TSC is an unsigned 64-bit counter that is incremented at every tick of the processor's internal clock. The implementation of the *entry* and *exit* hooks samples the processor's TSC, which on x86 architectures is a 64-bit unsigned integer containing the number of Central Processing Unit (CPU) clock ticks since it was last reset. The TSC is set to zero during the initialization of the processor or by request of the system administrator, by issuing a TSC reset routine. At every CPU clock cycle, the TSC is incremented.

The TSC can be sampled by issuing either of the following instructions: `rdtsc`[7] or `rdtscp`[8]. Both instructions store the current value of the TSC into the registers pair `EDX:EAX`, with the high-order 32-bit value of the TSC stored in register `EDX`, and the low-order in `EAX`. The `rdtscp` also stores the contents of the Model-specific Register (MSR) register `TSC_AUX` in the `ECX` register, which is the processor identifier. This identifier consists of an arbitrary 32-bit identifier, which is normally set by the operating system during startup or by the system administrator.

The practical difference is that `rdtscp` is intended to be used on multi-core environments because each core has its own TSC register. In such environments, it is important to know on which core the TSC was actually sampled to avoid imprecision due to *TSC drift*, a frequent condition where the cores operate at slightly different frequencies, thus incrementing the respective TSC at different rates. The *exit hook* implementation verifies whether the TSC difference is consistent by comparing the processor identifiers. In the case of a mismatch, the

[6]A common alternative approach is to sample the operating system's timer; however, such an approach is only valid when measuring events in the order of milliseconds or microseconds, depending on the resolution of the timers exported to user space.

[7]Mnemonic for "read time stamp counter".

[8]Mnemonic for "read time stamp counter and processor identifier".

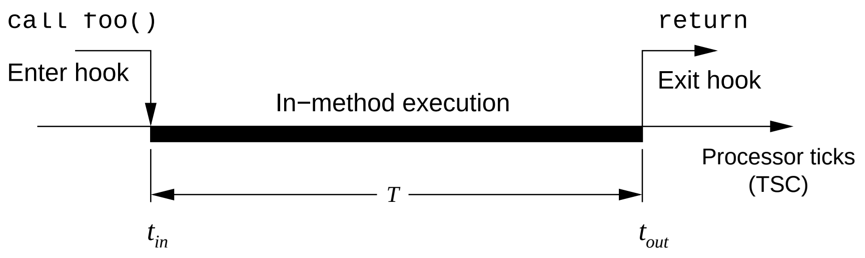

Figure 4.2: Measuring time spent in a method by means of instrumentation. When the method `foo()` is called, it triggers the entry hook that takes an initial timestamp t_{in} of the processor's TSC. When the method returns to the callee, it triggers the exit hook that takes a second timestamp, t_{out}, and the time spent in the method T is updated in its instrumentation data block.

measurement is discarded and the invocation count is not incremented. While it is possible to synchronize the TSCs across multiple cores, this is not practical and the overhead of doing so is significant, mostly because of the frequency that such synchronization must be repeated. Moreover, it is not uncommon for the operating system to rely on the TSC as a time source, thus synchronization can be disruptive.

The rate at which core-switching occurs depends on the load-balancing policies of the operating system. In Linux, the load balancer can migrate threads roughly once every 200 ms.[28] In practice, load balancing may occur once every few seconds (≤ 10 s). Without the core identifier, the instrumentation code can only detect a core switch if the measurement reports a negative time spent in the method.[9]

Figure 4.2 illustrates how the time measurement takes place. When the method `foo()` is invoked, it triggers the enter hook which takes an initial timestamp (t_{in}). After the method completes execution, any of the exit hooks are triggered when the method returns to the callee, taking the final timestamp t_{out}. The execution time $T = t_{out} - t_{in}$ for this invocation is updated in the instrumentation data block of the method, and the invocation counter is incremented. In multi-core environments, the exit hook verifies whether t_{out} was taken in the same core as t_{in}. The core identifier for timestamp t_{in} is also saved in the instrumentation data block. If they differ, the measurement T is discarded and the invocation counter of method `foo()` is not updated, avoiding an inconsistent sample.

The instrumentation mechanism performs an additional type of work: sending recompilation requests of the instrumented method to the JVM in order to quickly explore different compilation-plan modifiers. A recompilation threshold is computed during the first eight invocations of the method when it is compiled for the first time. This threshold, which can vary between 50 and $50,000$, determines how many method invocations will take place

[9]If the measured time spent in the method is positive, the clock frequency on the core that executed the exit hook is *higher* than the clock of the core that executed the enter hook. However, in this case the instrumentation code cannot detect that the measurement is incorrect.

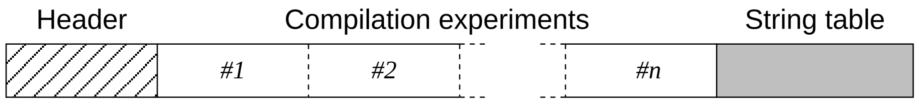

Figure 4.3: Organization of the binary archives. Every archive contains a header (hatched area) containing identification information such as revision and size of internal records, followed by a series of n experiment records. The archive is terminated by a string table (grayed area), that spans after the last experiment record to the end of the file.

between two consecutive compilations of a given method.

The objective is to allow the method to accumulate the equivalent of 10 ms of running time between compilations. Short-running methods tend to get a recompilation threshold close to $50,000$ invocations, while longer-running methods tend to get a threshold closer to 50. With this mechanism, methods quickly explore a large number of compilation-plan modifiers.

When the data collection is completed (*i.e.*, the application being executed under data-collection mode finishes), the collected data is stored in an archive, as detailed in Section 4.5.

4.5 Storing Collected Data

When Testarossa is executed in data-collection mode, it stores all collected data in temporary data structures during its execution to avoid the interference that would be otherwise caused by I/O operations (*e.g.*: updating logs as each compilation completes). After the application completes its execution, the data collected is stored in a binary archive for future use when training a machine-learned model.

The internal organization of these binary archives is illustrated in Figure 4.3. At the beginning of every archive lies a header (hatched area) containing a diverse set of information regarding the archive. The header includes details of the archive, such as format revision and the number of records. The header is especially important for reader applications, in order to verify whether they can decode the archive correctly. A series of the n *compilation experiments* recorded follows the header. A compilation experiment records the features of a method, the compilation-plan modifier used, and the associated measurements (*i.e.*, compilation time, execution time, and invocation count). After the last record, a string table spans until the end of the file, storing the signatures of the methods for the experiments recorded.

The components of the archive are detailed in the following sections, starting with the header in Section 4.5.1.

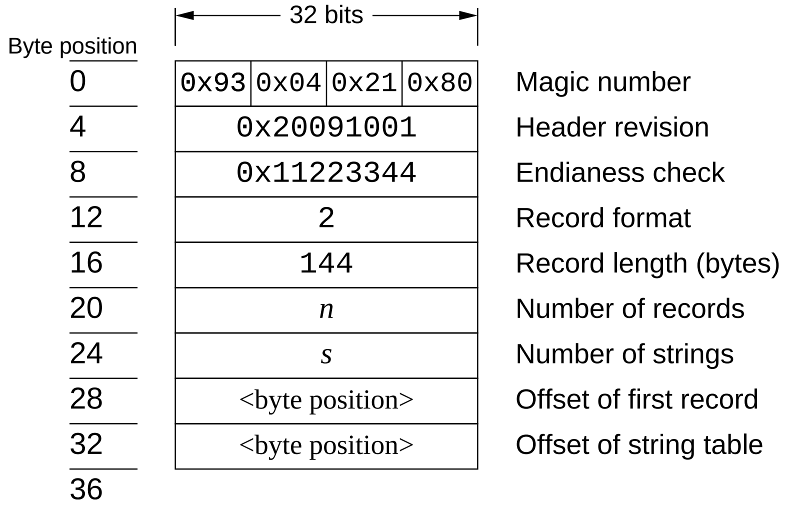

Figure 4.4: Binary archive header format. The header lies in the very first byte of the archive file, and consists of nine 32-bit words. The rightmost part of the figure briefly describe each of the words in the header, with the leftmost number detailing the respective byte position in the file.

4.5.1 Archive Header

Binary archives generated after data collection contain a header that provides reader applications that the information needed to verify that they will be able to continue decoding the archive. The header always starts at the very first byte of the file.

Figure 4.4 illustrates the format of the header. The header consists of nine 32-bit words, shown in the central part of the figure. The only exception is the magic number, which is laid out as a sequence of bytes to avoid endianess issues.[10] A brief description of each of those words appears on the right side of the figure; the left side contains the byte position of each word in the header. The purpose of each word in the header is detailed as follows:

Magic number An arbitrary numerical identifier, defined as the bytes `0x93`, `0x04`, `0x21`, and `0x80` in succession, forming the hexadecimal value `0x93042180`. They are laid at the very first byte in the file as individual bytes, for an endian-neutral representation.

Header revision The revision date of the header format, encoded as a hexadecimal value in the form `0xYYYYMMDD`, where `YYYY` corresponds to the year, `MM` to the month, and `DD` the day of the revision. It is only changed when the header format changes, allowing applications to either support loading archives in an older format or to detect the case where they cannot properly decode the archive, requiring an update. At the time of

[10]Endianess is the order in which the underlying architecture represents multi-byte values: *big-endian* architectures (*e.g.*: MIPS, PowerPC) store the most significant byte first in a multi-byte word; *little-endian* architectures (*e.g.*: Intel) store the least significant byte first. Some architectures are considered *bi-endian* (*e.g.*: ARM), allowing the endianess to be changed while an application is executing. Other conventions are referred to as either *mixed-endian* or *middle-endian*. Network byte-order is a synonym for big-endian.

this writing, the revision is `0x20091001`, meaning that the format was last updated on October 1, 2009.

Endianess check An arbitrary numerical identifier allowing the application handling the archive to detect an endianess mismatch. It is defined as the hexadecimal value `0x11223344`. If an endianess mismatch occurs, the reader application can continue decoding the archive if it can convert the byte order of the records to the appropriate format.

Record format The numerical revision of the format for the records storing compilation experiments. It is incremented by one each time the format changes, allowing applications to either handle archives with data in previous format or to detect when they are not able to decode the archive, requiring an update. At the time of this writing, the latest format revision is 2. The records are detailed in Section 4.5.2

Record length The length of each individual record, in bytes.

Number of records The number of records present in the archive.

Number of strings The number of textual strings in the string table that stores method names.

Offset of first record The byte position in the file where the first experiment record is located.

Offset of string table The byte position in the file where the string table for method names is located. From its starting location, the string table extends to the end of the file.

The sequence of records follows the archive header. Section 4.5.2 describes the format of such records.

4.5.2 Experiment Records

Every experiment performed in Testarossa is stored in data structures in memory until the application terminates. When the application terminates, the records generated by the experiment are written to disk in the same format as the structures stored in memory. The simplicity of this design allows for a very straightforward implementation and keeps the binary archives small. Each record has a fixed length of 144 bytes. These records store all information collected by a single compilation experiment.

Figure 4.5 illustrates the format of the data structure for each experiment record. Around the figure there is a byte legend. Each row is 8-bytes wide, with a marking for every 8-th byte boundary in the legends on the left and right sides of the figure. Within each row, byte

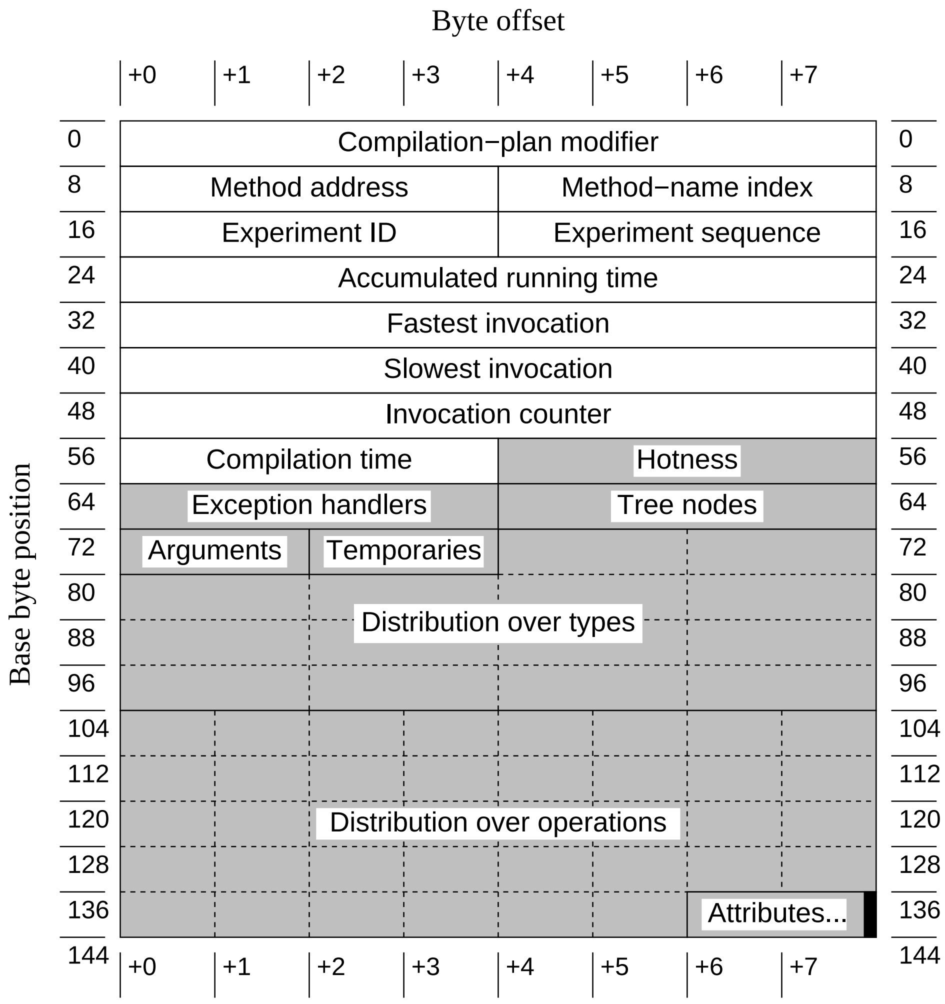

Figure 4.5: Binary archive record format. Each record carry information regarding a single experiment (white area) and the feature vector $\vec{F}$ for the method (gray area). To avoid alignment issues, the record is padded out to a 64-bit boundary (dark area).

offsets are highlighted at each byte boundary and marked in the upper and lower legends. For example, the field “Node count” lies at the 64-th byte row with a byte offset of +4, meaning that it is located at byte $64 + 4 = 68$ in the data structure, spanning bytes 68–71.

The data structure is composed of (a) information pertinent to a compilation experiment (white area), which contains the compilation-plan modifier used and the running time measured, and (b) the feature vector $\vec{F}$ for the method used in the experiment (gray area). The format also contains padding (dark area) to ensure data alignment at 64-bit boundaries. This padding ensures that the same data structure can be used either for memory-based (during data collection) or disk-based operations (when storing/loading records from an archive).

Each field of the data format is detailed as follows.

Compilation-plan modifier Stores the compilation-plan modifier used in the experiment as a stream of N bits, where N is the number of transformations that can be selectively disabled. The i-th bit in the stream controls the i-th transformation supported. The transformations controlled are detailed in Chapter 5.

Method address For debugging purposes, the record contains the starting address of the method while Testarossa was executing, so that it can be cross-referenced from other logs if necessary.

Method-name index Indicates the index for the method signature in the string table.

Experiment ID Stores the identifier for a set of experiments E, allowing reader applications to locate experiments performed on the same method.

Experiment sequence The sequence number for the experiment described in the record, within the set of experiments E.

Accumulated running time The accumulated running time $\sum_{i=1}^{I} T_i$, where i is the i-th invocation of the method from a total of I invocations recorded, and T_i is the running time of the i-th invocation.

Fastest invocation The amount of CPU cycles spent in the fastest invocation of the method.

Slowest invocation The amount of CPU cycles spent in the slowest invocation of the method.

Invocation counter The number of invocations recorded for the version of the method in the experiment record.

Compilation time The compilation time for the experiment, in microseconds.

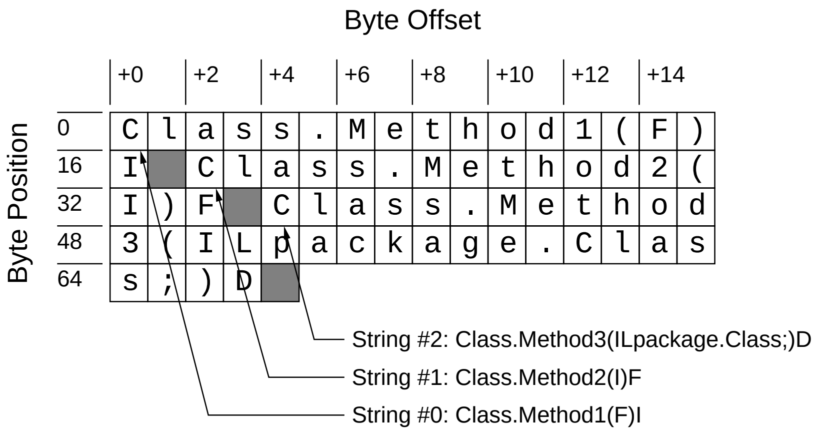
Byte Offset
+0
+2
+4
+6
+8
+10
+12
+14
Byte Position
0
16
32
48
64
C l a s s . M e t h o d 1 (F)
I C l a s s . M e t h o d 2 (
I) F C l a s s . M e t h o d
3 (I L p a c k a g e . C l a s
s ;) D
String #2: Class.Method3(ILpackage.Class;)D
String #1: Class.Method2(I)F
String #0: Class.Method1(F)I

Figure 4.6: String table format. The string table stores a total of S unique method signatures as null-terminated strings (the null character is displayed as a darkened box). The index of the first string is 0, and the last string is $S - 1$. The strings are stored in arbitrary order.

Hotness The optimization level (hotness) as decided by Testarossa.

Exception handlers The number of exception handlers in the method.

Node count The number of nodes in the intermediate representation tree for the method during compilation, used by Testarossa.

Arguments The number of arguments that the method receives when invoked.

Temporaries The number of temporaries used in the method.

Distribution over types The counters comprising the distribution over types (detailed in Section 4.2.2).

Distribution over operations The counters comprising the distribution over operations (also detailed in Section 4.2.2).

Attributes The set of 15 binary attributes for the method (detailed in Section 4.2.1).

In the archives, a string table containing the signatures of the methods is placed after the last recorded experiment, and is detailed in Section 4.5.3.

4.5.3 String Table for Method Names

The last data block in a binary archive is the string table, containing the signatures of the methods with recorded experiments in the archive. The string table is useful when diagnosing issues or when cross-referencing compilations across different logs. It has no effect on the machine-learned models currently in use.

Figure 4.6 illustrates the format of the string table. A string table contains S unique

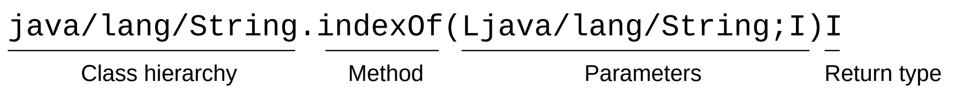

Figure 4.7: Signature of Java methods, composed of a class hierarchy, a method name, zero or more parameters expected by the method, and a return type.

Table 4.4: Signature of Java methods, composed of a class hierarchy, a method name, zero or more parameters expected by the method, and a return type.

Character	**Java type**	**Remarks**
`B`	`byte`	
`C`	`char`	
`D`	`double`	
`F`	`float`	
`I`	`int`	
`J`	`long`	
`S`	`short`	
`Z`	`boolean`	
`V`	`void`	Only valid for return types
`L`*class*`;`	Class	Parameter or return type is an instance of *class*
`[`*type*	Array	Array of type *type*

strings, where each string has an associated index. The first string in the table has index 0, while the last one has index $S - 1$. During data collection, the string table is updated dynamically as index lookups are performed. A lookup returns either a fresh index if the signature is unique, or an existing index if a same signature was inserted in the table previously. When the application under data collection completes execution, the string table is saved, with the strings in the order in which they were inserted in the table.

Strings in the string table represent method signatures in accordance to the Java Virtual Machine Specification [17]. A typical method signature is illustrated in Figure 4.7. A method signature starts with the fully-qualified class hierarchy, followed by the method name. The usual dot-delimited syntax for programs in Java syntax is replaced by slashes, and a dot separates the hierarchy from the method name in the signature. After the method name, the parameters expected by the method are specified in a character-based format, followed by the return type of the method using the same convention. The specification is left blank for the parameters part if the method does not expect parameters, but the parentheses must be included regardless.

The character-based encoding for specifying parameters and return types is illustrated in Table 4.4. The primitive types are all represented by single, different characters (*e.g.*: character `B` for a `boolean` parameter or return type). If the parameter or return type is a class, the class is specified as a class hierarchy enclosed by characters `L` and `;` (*e.g.*: `Ljava/lang/String;` for a parameter or return type of `java.lang.String`).

Arrays are specified using the `[` character, depending on the number of dimensions of the array. For instance, the double-dimensional array `int arr[][]` becomes `[[I` when specified with the character-based encoding. If the method does not return a value, its specification is `V` (after *void*).

Section 4.6 presents the motivation for a customized infrastructure in Testarossa, from profiling to binary archiving format.

4.6 Motivation for a Customized Infrastructure

The main reason for using a customized profiling infrastructure is the failure of the available solutions to completely fulfill all requirements of the data-collection process, namely:

1. Low execution overhead, to reduce JVM execution time, which can limit the number of experiments that can be performed within a time budget.

2. Sensible storage requirements, because a very large number of compilation plans must be tested.

3. High-resolution measurements, because, in practice, many method invocations execute for a very short amount of time.

Most available solutions tackle either #1 and/or #3. The number of compilation experiments that must be performed to generate sufficient data tends to be very large, especially because the many experiments are performed in a single JVM execution. Thus, developing a customized profiling infrastructure that meets all three requirements is a must.

Profiling alternatives are discussed in Section 4.6.1.

4.6.1 Alternatives for Measuring Time

Time measurement based on TSC is an attractive solution. These counters are widely available, being supported even on some non-x86 architectures, and combine high-resolution measurements with simple usage and low execution overhead. Measuring events with TSC as a reference is a matter of issuing the appropriate instruction to read from the TSC.

Existing solutions were analyzed in the early stages of the project, and did not meet the requirements, either due to excessive overhead, complexity, or reduced availability. Two prominent alternatives were evaluated: system timers and profiling suites.

System Timers

The simplest approach to measuring time is to use the facilities provided by the underlying operating system. On UNIX-like operating systems, such as GNU/Linux, `gettimeofday()` is the common choice. The advantages of using `gettimeofday()` are twofold: (*a*) issues related to multi-core environments are handled by the operating system; and (*b*) the extra time imposed by each call is often less than 250 CPU cycles on recent versions of the Linux kernel.

The resolution provided by `gettimeofday()` is, however, too coarse, being limited to the microsecond scale. In practice, method invocations accounting for $2,000$ CPU cycles or less are very frequent. On processors clocked at 2 GHz, $2,000$ CPU cycles translate to 1 µs, and any invocation consuming less than $2,000$ CPU cycles on those system cannot be reliably measured by `gettimeofday()`.

A more elaborate approach is to offload the event measuring to the hardware, if this feature is supported. Performance Monitoring Counters (PMCs) were introduced in the Intel Pentium architecture, and provide a diverse range of performance events that can be monitored directly by the underlying hardware, without much of the overhead.

The main drawback is that PMCs are not always available. Either they are not supported by the processor, or their use is not allowed by system administrators. In addition, PMCs contain processor-specific events that may be supported in one platform but not be available on another. Often the operating system does not grant access to PMC infrastructure by default, requiring the system administrator to explicitly enable it on a system-wide basis or for specific applications.

The workflow when using PMCs is to select the events of interest and signal the processor by writing the request to a MSR to start collecting data for the events selected [2]. After the operation is complete (*e.g.*: a method invocation completes), the processor is signaled to stop collecting data, or to provide a snapshot of the events being monitored. The actual implementation and the limits on the number of parallel events that can be offloaded to hardware are processor-specific.

As PMCs become more widely available and standardized across different platforms, a PMC-based approach will be very effective. The next section presents a software-based alternative that provides simple-to-use high-resolution counters.

Dpiperf

Dpiperf [23] is part of a suite of performance-monitoring tools from IBM. Dpiperf offers off-the-shelf solutions for most performance-tracking tasks and it has special support for Java and C/C++ languages.

The suite offers high-resolution timers and very fine-grained control over events. The high-resolution timers use a kernel driver (supported in both Linux and Windows environments), allowing applications to capture time events on a per-thread basis. Dpiperf integrates with the Java Virtual Machine, allowing events to be generated from within the JVM without disrupting the execution of the application.

Despite providing the desired high-resolution measurements and a detailed picture of the application runtime behavior, Dpiperf induced an overwhelming overhead during experimental profiling attempts. The overhead is two-fold: (*i*) execution overhead, resulting in execution slowdowns in the application being profiled of 500× on average; and (*ii*) storage overhead, generating logs that are several gigabytes long.

Measuring method-execution times using Dpiperf generates a considerable amount of communication between the JVM and the kernel driver (which ultimately sets up the events of interest and reports the results when an event completes). The native format of the logs generated by Dpiperf is text-based and requires a post-processing stage to extract information equivalent to that available in the experiment records discussed in the previous section. Text-based logs are often bulkier than binary logs and grow very quickly because most applications make thousands of method calls during their lifetime.[11]

During the evaluation of Dpiperf, an additional layer was implemented in Dpiperf to intercept the log generation and provide on-the-fly compression based on `zlib` [29], which is available on most operating systems. The extension achieved compression rates in the order of 90–95% and reduced the execution slowdown to 50×, on average.

While the level of detail and timing resolution were suitable, the overhead both in terms of execution time and storage was not and a Dpiperf-based approach was abandoned in favor of a leaner solution.

As was the case with profiling, alternatives for storage of collected data were evaluated and are discussed in Section 4.6.2.

4.6.2 Alternatives for Storing Collected Data

The main reason for a customized storage format comes from the need to generate a large amount of data for the learning process, and to avoid creating unnecessary dependencies in Testarossa. To do so, applications used for data collection must be executed for long periods and repeated numerous times. Each execution generates thousands of compilation experiments, depending on the application.

The simplest alternative is to output the data in a text-based format, possibly formatted in a structured format such as Extensible Markup Language (XML). The advantage of such an approach is that the output is, to some extent, human-friendly. However, due to the amount of data collected, the resulting logs can easily grow to several megabytes. For example, the feature vector alone formatted in the textual format required by the machine-learned models consumes from 500 to $1,000$ characters. A regular data-collection execution generates from $4,000$ to $15,000$ compilation experiments, resulting in $\frac{500+1,000}{2} \times$

[11]During the evaluation, logs frequently grew past 4 GiB in size, demanding considerable time for compression and post-processing.

$\frac{4{,}000+15{,}000}{2} \approx 6.8$ MiB worth of data, only for the feature vectors, and for a single data collection.

Alternatives based on databases are attractive because they provide a wide selection of features, especially Berkeley DB[12] and equivalent implementations (GNU DBM). These databases support advanced features such as efficient locking mechanisms, concurrent access, and large storage capacity.

The disadvantages of using such databases include both portability and licensing issues. Because they add external dependencies to the data-collection process attached to Testarossa, they can limit the platforms to those that are supported by the databases. Moreover, Testarossa would then be bound by the licensing conditions of these databases.

4.7 Concluding Remarks

The data-collection process changes the usual behavior of Testarossa, with the goal of exploring different compilation plans. The hypothesis is that, for a number of methods, alternative compilation plans may perform better than the original compilation plans in Testarossa. The ability to efficiently store collected data, either while the compiler is running or after the data-collection process is complete, is a requirement for large-scale data collection.

The key component of data collection is the generation of compilation-plan modifiers. Chapter 5 details the compilation-plan modifiers and the approaches used to generate them.

[12]Berkeley DB is an embeddable database system with only local access (no networking support) by means of Application Programming Interface (API), currently maintained by Oracle Corporation.

Chapter 5

Compilation-plan Modifiers

The generation of compilation-plan modifiers is a key operation for data collection. The approach used to generate such modifiers guides the compilation experiments in different ways. Currently, the two methods implemented are: (*i*) a pure randomized search, that explores different compilation-plans very aggressively (detailed in Section 5.3); and (*ii*) a progressive randomized search, that generate modifiers that gradually diverge from the original optimization plan used by Testarossa (Section 5.4).

Before detailing the generation approaches, the compilation-plan modifier itself is described in Section 5.1.

5.1 A Compilation-plan Modifier

A compilation-plan modifier is a stream of bits that either enables or disables a specific transformation in Testarossa, as illustrated in Figure 5.1. The plan is L-bits wide, where L is the total number of transformations controlled by the machine-learned model, with $L = 58$ at the time of this writing. Thus, the i-th transformation is either enabled or disabled depending on the value of the i-th bit in the compilation-plan modifier. The bit assignments are zero-based, therefore the first transformation is assigned bit 0, and the

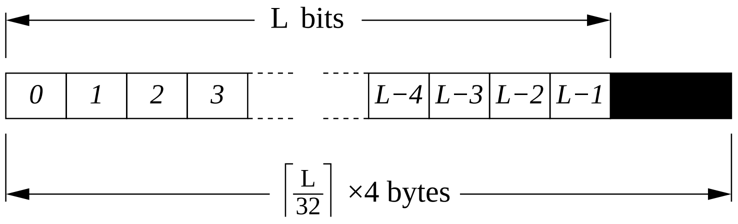

Figure 5.1: Representation of a compilation-plan modifier, as a stream of L bits. The i-th bit in the stream corresponds to the i-th transformation that can be controlled by the machine-learned model in Testarossa. When stored, a compilation-plan modifier uses $\lceil \frac{L}{32} \rceil \times 4$ bytes to avoid alignment issues.

$(L-1)$-th bit controls the last transformation. When stored in memory or archives, the modifier takes $\lceil \frac{L}{32} \rceil \times 4$ bytes to avoid alignment issues (8 bytes for $L = 58$).

A compilation-plan modifier is used during the optimization stage of Testarossa. When Testarossa makes a decision to apply or not apply a transformation, it takes an additional vote that is cast by the **strategy control** module. This module is discussed in detail later, in Chapter 7. For now it suffices to know that this module is queried with a transformation identifier internally used in Testarossa. This identifier is translated to the appropriate bit position, and returns the recommended action (to apply the transformation or not). If the transformation is not controlled by the module, the module casts a blank vote that does not affect the decision made by Testarossa.

The vote cast by the module is not a final decision. Testarossa has an intricate mechanism to decide not only whether or not to apply a transformation, but also if it is possible to apply it. For example, if for some reason the strategy control module casts a positive vote on a loop transformation but the method does not have loops (as determined by Testarossa), Testarossa will not apply the transformation. Forcing Testarossa to apply such a transformation in this case would lead the compiler to crash.

A compilation-plan modifier does not change the order in which the transformations are applied. When Testarossa is running in production mode with support from the machine-learned model, it still selects the optimization level for a method being compiled. The modifier only disables certain transformations in the optimization plan.

There are multiple approaches for generating compilation-plan modifiers. Section 5.2 discusses the characteristics common to these approaches.

5.2 Overview of the Exploration of Compilation Plan Modifiers

When Testarossa executes in data collection mode, M compilation plan modifiers are pre-computed, for each of the optimization levels in Testarossa. This pre-computation happens before data collection starts. The value of M depends on the expected number of compilations and on the expiration rate of the compilation-plan modifiers. For example, the value used in data collection for this study was set to $2,000$ compilation-plan modifiers.

The value of M needs to strike a balance and has to be set by the developer prior to the start of the data collection process. On one hand, a large M produces more than enough compilation-plan modifiers, allowing frequent methods to be recompiled several times and for the exploration of alternative plans. On the other hand, a pool of compilation-plan modifiers much larger than necessary delays the beginning of the data collection process. In practice, the value can be quickly determined by experimentation across the set of applications to be used for data collection.

The expiration rate of compilation-plan modifiers defines an upper bound on the number of compilations that are performed using a modifier, and is a tuning parameter for the strategy control, depending on the aggressiveness of the exploration desired. After a modifier is expired, it is no longer used. A high expiration rate means that a modifier is used to compile only a few methods before it is no longer used during the same JVM run. If the expiration rate is sufficiently high, a data collection experiment may exhaust the pool of pre-computed compilation-plan modifiers. At that moment, the methods in the application gradually reach a “no recompilation” state and eventually the application progresses normally.

Another characteristic that is common across the approaches of generating compilation plan modifiers is to reserve a special modifier. The **null modifier**, as it is called, is special for two reasons: (*i*) it never expires and (*ii*) it does not change the original compilation plan in any way (thus its designation). This special modifier is encoded with all bits set to 1. Depending on the approach of generate modifiers, the null modifier may be used very early in compilation experiments for each method, or later.

The advantage of using the null modifier early (*i.e.*, as the first modifier) is to ensure that all methods will be compiled using the original plan in Testarossa. This plan has been hand-tuned over the course of many years by expert developers. However, for infrequently compiled methods it could be the only plan used, even across multiple JVM runs. On the other hand, having the null modifier later in the data collection process can also be advantageous. If a method is infrequently compiled across multiple JVM runs and does not account for a significant portion of the execution time of the application, it is possible that such a method will not benefit from the original compilation plan in Testarossa.

Section 5.3 discusses the first approach to generate compilation-plan modifiers, which is randomized search.

5.3 Randomized Search

The generation of compilation-plan modifiers using randomized search is the simplest form of exploration. In this approach, M modifiers are generated using the random number generator available in the operating system, for each optimization level supported by Testarossa.

Randomized search can be considered aggressive in the sense that the effective compilation-plan (after the modifier is applied) is very different from the original plan (*i.e.*, on average, those plans will have more bits disabled than plans generated with approaches that start with the null plan and modifies it slowly). Thus, the random-search approach can quickly sample the compilation-plan modifier space, which can be advantageous if the goal is to find plans that are much cheaper in terms of compilation cost.

The null modifier is the third modifier used. The hypothesis is that some of the methods that are not recompiled multiple times during data collection are infrequent and, thus, might not benefit from the original compilation plan used by Testarossa. Conversely, a frequent method is more likely to benefit from the original optimization plan and thus will be recompiled enough times to eventually use the null modifier.

The expiration rate of the randomized search is set to 50 compilations of distinct methods. This rate produces a slower expiration rate than the other generation approaches currently implemented. This slower rate is intentional because the modifiers generated by the randomized search are very different from each other.

An alternative approach to generate modifiers complementary to the randomized search just discussed, is presented in Section 5.4.

5.4 Progressive Randomized Search

A progressive randomized search can generate compilation-plans in a random fashion, but with a strong, controlled bias. The idea is that each transformation starts with a null initial chance of being disabled, but at each round this chance grows by a small rate (mutation) defined by

$$D_i = i \times \frac{0.25}{L}, \quad 0 \leq i \leq L. \tag{5.1}$$

The first round ($i = 0$) generates the null modifier. Moreover, any subsequent i-th round ($i > 0$) is guaranteed to disable at least one transformation because every modifier must be unique. A method being compiled for the i-th time (where $i = 0$ is a first-time compilation) uses the i-th modifier.

In Equation 5.1, D_i is the probability that any transformation will be disabled in the i-th round of modifier generation, *i.e.*, each transformation is individually evaluated to decide whether it will or will not be disabled. In the first round ($i = 0$), the chance is $D_0 = 0$. This chance grows until $D_L = 0.25$ (25%). The probability D_i increases by 0.000125 at each subsequent round.

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 (and, to some extent, slowly) diverge from the original compilation-plan in Testarossa. In this approach, 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 ideal for many cases, with a number of other cases requiring small tune-ups that can benefit from a method-specific approach. In practice, modifiers that diverge too much from the original compilation plan (*e.g.*: 50% or more transformations disabled) tend to exhibit poor performance.

5.5 Considerations on Exploration Approaches

Theoretically, *each* distinct feature vector representing a distinct method[1] has a space of $2^L = 2^{58} \approx 2.88 \times 10^{17}$ possible modifiers to explore. In practice, most of these modifiers will not create better optimization plans. This observation suggests a more elaborate search based on heuristics. Such heuristic-based search requires feedback during the data-collection process, allowing it to focus the search on regions within the space of possible modifiers.

The randomized search can sample the full space of modifiers quickly, whereas the progressive randomized search focuses on alternatives similar to the original compilation plan in Testarossa. If an alternative plan for a method diverges considerably from the original plan in Testarossa, the randomized search has better chances of finding it than the progressive approach. Conversely, the progressive search has better chances of finding modifiers that tune the original plan in Testarossa to better suit a method.

Fortunately, one does not have to choose between these exploration approaches unless the time available is too restricted. The outcomes of both exploration approaches can be combined to increase the chances that an alternative optimization plan is found. In any case, it is important to include the original compilation plan from Testarossa, represented by the null modifier. If the compilation plans explored are of inferior quality, the machine-learned model will still be able to use the original plan when it is the best one.

When creating a final data set from several data collections, one has to rank these sets to retain the modifiers that performed better during the exploration. The ranking process used in this study is detailed in Chapter 6.

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

Chapter 6

Learning a Model

Training a machine-learned model involves processing the data collected from applications when running Testarossa in data collection mode, and selecting appropriate parameters for the SVM.

The data processing is broken into two phases: (*i*) processing itself, by unarchiving the data and ranking it to produce a data set; and (*ii*) fine-tuning the resulting 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.

The initial step, the preparation of data sets, is discussed in Section 6.1.

6.1 Preparing Data Sets

The preparation of a data set involves several processes: (*i*) unarchiving collected data into intermediate data sets; (*ii*) optional merging of intermediate data sets; (*iii*) ranking the data, and then generating a final data set.

While archived data is very compact, unarchiving it into an intermediate format brings the advantages of using tools that are usually shipped with the machine-learning algorithm implementations. In the case where these tools are not sufficient, customized tools can be implemented to handle the data and even interface with some of these third-party tools. This was initially the case in this study, until the performance became a concern when creating large data sets from an even larger pool of data.

The merging of intermediate data sets, while optional, is also important in the training of machine-learned models. The key advantage is being able to prepare independent intermediate data sets, and then selectively merging the data sets of interest. One common technique for evaluating machine-learned models, *cross-validation*, can be easily carried out with this approach. Cross-validation consists on setting aside portions of the available data, training the model and testing it on the data set aside.

A variation of this technique is called *leave-one-out cross-validation*. In this variation, a pool of P applications are used for data collection. When creating the training data sets, P data sets are created, each using data from $P-1$ applications, and the test is performed on the application whose data had been set aside. Data sets for leave-one-out cross-validation are easily produced by merging the appropriate intermediate data sets.

Finally, the final data set is created by ranking the intermediate data set (or merged intermediate data sets). The ranking process evaluates the quality of the compilation-plan modifiers used in experiments, and allows for the selection of the best-performing plans under different policies.

The first process, unarchiving of data, is detailed in Section 6.1.1.

6.1.1 Unarchiving Data

The first step in preparing training data sets is to extract the information from the binary archives, which were generated during data collection, and to organize them in the format required by the subsequent tools. For this, a reader application was implemented, reusing the code that is built into Testarossa for data collection, specifically the code to handle compilation experiment records and binary archives. The result is saved as an **intermediate data set**, because it requires further processing before it can be used for the training of a model.

The reader application takes a binary archive as input, validates it, and extracts the data. During the validation, the reader verifies that it can actually decode the archive (by inspecting the main header in the archive and testing for endianess issues). If the validation passes, the reader will process each of the available records.

Data extraction is straight-forward, but requires collaboration with the subsequent tools. The reader application extracts the running and compilation times for a method, the compilation-plan modifier used for the compilation, and the associated feature vector, generating an intermediate data file. Figure 6.1 illustrates the intermediate format of such files. The column *Execution Time* contains the total running time for the respective method, in CPU cycles. The column *Compilation Time* has the compilation time for that experiment in microseconds, followed by the invocation count in the column *Invocations*. The column *Modifier* contains the compilation plan modifier in decimal representation. Lastly, the column *Feature Vector* represents the feature vector for the method. The feature vector in the figure is only included symbolically because it is very long when represented as text, but it suffices to know that each component is represented in decimal representation, and separated by a single space character. In the figure there are only two distinct methods exemplified; one appearing in two records and the other in three records. Each record represents a different compilation of the method.

Execution Time (CPU cycles)	**Compilation Time (μs)**	**Number of Invocations**	**Compilation-plan Modifier**	**Feature Vector**
21,491	207,018	8	`0x03ffffffffffffff`	$\vec{F}_{\text{method A}}$
31,345	324,192	62	`0x03fdffffffbffdff`	$\vec{F}_{\text{method A}}$
10,271,285	131,366	4,822	`0x03edffbfedfffff7`	$\vec{F}_{\text{method B}}$
12,636,564	128,734	25,296	`0x03fdffffedfffffd`	$\vec{F}_{\text{method B}}$
29,814	324,950	912	`0x03ffffffffffffff`	$\vec{F}_{\text{method C}}$

Figure 6.1: Contents of an intermediate data set. The first column contains the total running time for a method after a given compilation in CPU cycles, followed by the compilation time in microseconds and the invocation counter. Next are shown the compilation plan modifier in hexadecimal representation and the feature vector for the method. The intermediate data set is in text format, and each line of the file represents a compilation experiment record.

The format of the file is a textual tabular representation, with one record for each line of the file. This allows for easy inspection of the data, as well as for manipulation with a wide range of text-manipulation tools (`grep`, `awk`, and others). In addition, the intermediate format permits partial data sets to be merged in arbitrary combinations, for example, by having one partial data set for each application submitted to data collection and merging data from applications of interest.

With the intermediate data set generated, its data can be ranked, as detailed in Section 6.1.2.

6.1.2 Ranking Data

The ranking process takes a number of intermediate data sets as input and generates a final data set that can be used to train a machine-learned model. A customized application was implemented to perform the ranking over reasonably large amounts of data. The application is called `feat2svm`, named after the process "feature vectors to SVM format", and is implemented in the C++ programming language.

Figure 6.2 illustrates the ranking workflow 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 experiment. This sorting aggregates all experiments performed on the same feature vector. For each i-th record in the intermediate data sets, the objective ranking function

$$V_i = \frac{R_i}{I_i} + \frac{C_i}{T_h} \tag{6.1}$$

is computed, and the resulting value V_i is included with the respective compilation-plan modifier for the corresponding feature vector $\vec{F_i}$. V_i is the normalized cost for the i-th record, where a compilation plan with smaller values of V_i is better for a method characterized by the features in the record.

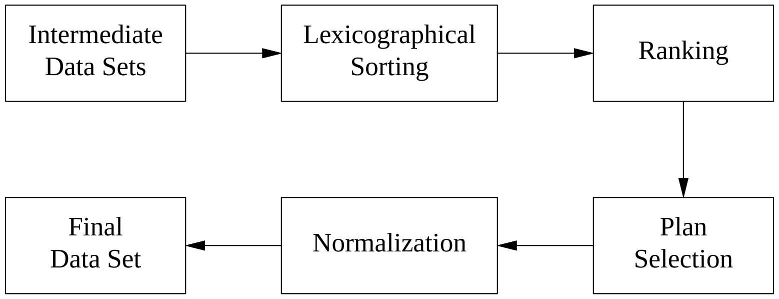

Figure 6.2: Intermediate data sets are combined and sorted lexicographically, aggregating compilation experiments performed on the same feature vector. The experiments are ranked to determine their relative quality, and the associated compilation plan modifiers from a subset of the experiments are selected. The results are normalized to improve numerical stability for the learning process, and a final data set is created.

In Equation 6.1, R_i is the total running time of the compilation using the respective modifier, I_i the invocation counter, C_i the compilation time, and T_h the triggering value used by Testarossa for recompiling at compilation level h (h is used to reflect the *hotness* or optimization level). So, $\frac{R_i}{I_i}$ is the average running time per invocation which should be minimized, and $\frac{C_i}{T_h}$ is the weighted compilation cost that also should be minimized. 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 is thought to contain many-iteration loops.

At the end of the ranking, each unique feature vector will have a set of pairs $\langle M_i, V_i \rangle$, where M_i is the modifier used in the i-th experiment, and V_i is the associated value as computed in Equation 6.1. The set of sorted experiments will have at least one of such pairs per unique feature vector.

The plan selection is specialized in three different ranking approaches to select compilation-plan modifiers for the final data set:

1. Single-best performing modifier

2. Top-N best performing modifiers

3. Top-$M\%$ best performing modifiers

The simplest approach is to select a single best performing compilation plan modifier for each unique feature vector, *i.e.*, the modifier associated with the lowest normalized cost V computed during the lexicographical sorting. It also results in smaller data sets since most of the experiments are rejected in favor of a single compilation plan modifier per unique feature vector.

The Top-N and Top-$M\%$ approaches are very similar. The former selects the N best performing modifiers for each unique feature vector. The latter, instead, selects all modifiers for each unique feature vector with a value V equal, or at most $M\%$ higher, than the best-performing modifier. For example, a Top-5% plan selection policy selects all modifiers where $V_0 \leq V_i \leq (V_0 \times 1.05)$. The range is recomputed for every feature vector. In addition, both Top-N and Top-$M\%$ approaches can be combined, resulting in a policy that selects at most N modifiers from the group of Top-$M\%$ performing modifiers for each unique feature vector.

After the modifiers are selected through the plan selection process, each component of the corresponding feature vector is normalized to the $[0, 1]$ range using

$$C_{\text{norm}} = \frac{C_j - C_{\min}}{\Delta C} \quad 0 \leq C_{\text{norm}} \leq 1, \tag{6.2}$$

where C_j is the j-th component of the feature vector, $C_{\min}$ is the minimum value seen during data processing, and ΔC is the difference $C_{\max} - C_{\min}$ (where $C_{\max}$ is the maximum value of the j-th component).

The normalization is not a mandatory step, but it greatly improves the learned models by eliminating the dominant effect of larger numerical ranges over smaller ones [24]. Besides, kernels, such as the d-degree polynomial kernel, can be affected by storage limitations when performing computations (*e.g.*: overflowing floating-point variables when raising large numbers to the d-th power). In practice, the collected data has significantly different numerical ranges over each component of the feature vector.

At the end of the normalization process a final data set is created. This data set is used to train a model, and its format is detailed in Section 6.1.3.

6.1.3 Data Set Format

The format of the final data obeys the rules required by LIBLINEAR, which is the format inherited from LIBSVM. 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 6.3 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 n components of the feature vector. If a component of the feature vector is zero, it can be omitted and the machine-learned model assumes it is zero. All non-zero components of the feature vector must be preceded by its component index, for example, `10:0.5625` denotes the value `0.5625` as the 10-th component of the feature vector for an arbitrary data instance.

Due to limitations imposed by the format, the class label can only express integer numbers in the range $[1, 2^{31}-1]$. Because the range used to represent modifiers is much larger, an alternative representation is used to generate the final data set. For now, it suffices to know

Class label	Feature vector components			
L_i	`1:`$F_{i,1}$	`2:`$F_{i,2}$	$\ldots$	`n:`$F_{i,n}$
L_{i+1}	`1:`$F_{i+1,1}$	`2:`$F_{i+1,2}$	$\ldots$	`n:`$F_{i+1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
L_N	`1:`$F_{i+1,1}$	`2:`$F_{i+1,2}$	$\ldots$	`n:`$F_{N,n}$

Figure 6.3: Data set format used by LIBLINEAR, where each line of the file is a data instance. Data instances are declared starting with the class label, followed by up to n components of the respective feature vector. Non-zero components must be explicitly prefixed with the component index, whereas zero-valued components can be omitted from the data set.

that the compilation plan modifier space is remapped into this smaller space supported by LIBLINEAR, without any adverse effects to the learning process. The remapping process is detailed in Chapter 7, including the procedure to map a class label back to a modifier.

At this point in the data-set preparation, the data set can be considered a final one and training of the machine-learned model can begin. However, if the size of the data set becomes a concern, an additional step, discussed in Section 6.2, can be added to the data-set generation phase to reduce its size.

6.2 Trimming

It is often the case that a final data set is too large for the timely training of a machine-learned model. In such cases, an additional application, `trimmer`, is used to remove data instances based on the frequency of the class labels. The application processes a final data set and, given a trimming factor C, outputs only classes with a frequency greater than or equal to C. For example, a trimming factor of 10 removes all data samples associated with classes that have 9 or less data samples in the data set.

Table 6.1 exemplifies the effects of different trimming factors in the size of a data set. Trimming a data set reduces its overall size by removing data samples with infrequent classes, which allows for more manageable training times if the data set is too large. A data set is considered too large if the training process demands so many resources that it interferes with itself and the rest of the system. For example, if the training application requests the majority of the available memory in the system, it will cause the operating system to swap pages excessively and, thus, will take much longer to complete.

In addition, the learning process may benefit from a smaller class-to-instances ratio and from a less-dominant presence of infrequent classes. Infrequent classes can overshadow more frequent classes due to their large overall number, with a possible negative effect in the learning process.

Figure 6.4 plots the effect of trimming on data-set sizes. The horizontal axis represents

Trimming Factor	Data Instances	Unique Classes	Class-to-instance Ratio
0	154,097	23,371	1:6.59
2	145,738	18,031	1:8.08
4	124,476	11,817	1:10.53
6	105,907	8,404	1:12.60
8	88,968	6,136	1:14.50
10	73,025	4,453	1:16.40
12	57,920	3,134	1:18.48
14	45,053	2,177	1:20.70
16	35,726	1,573	1:22.71
18	28,330	1,149	1:24.66
20	22,427	846	1:26.51

Table 6.1: Effect of different trimming factors in data sets. By removing infrequent classes (according to the trimming factor), the overall size of the data set decreases. In addition, removing infrequent classes helps rebalance the data set.

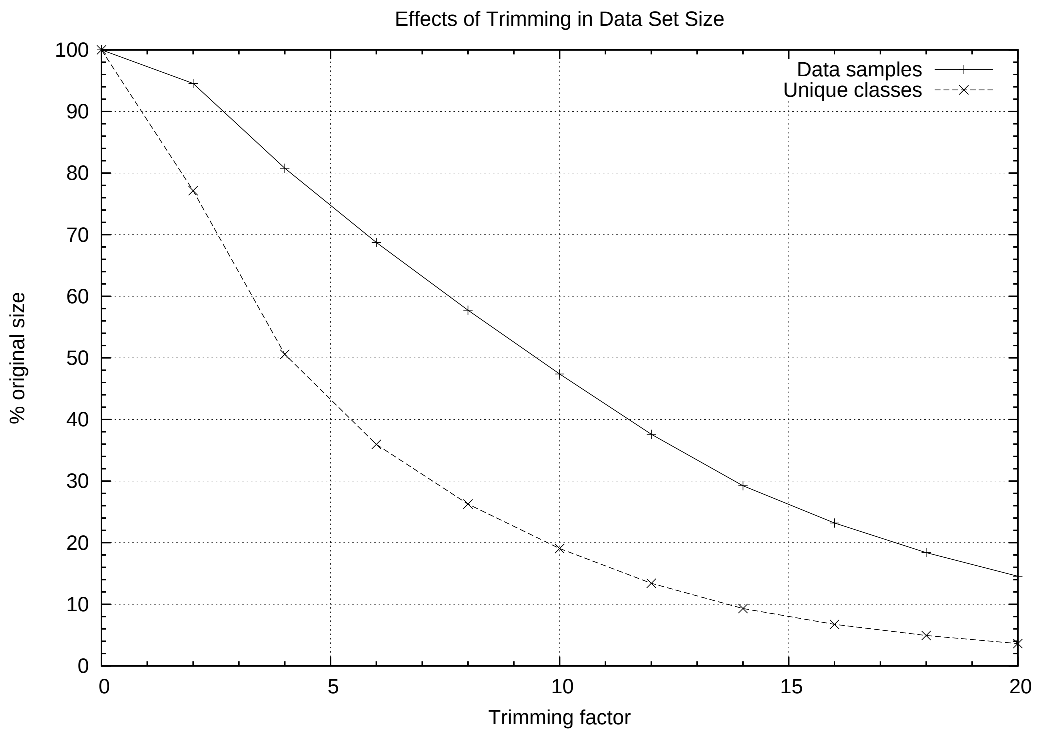

Figure 6.4: Trimming as a noise-filtering step. Even small trimming factors can remove a large amount of infrequent classes; at the same time the overall data set size is reduced.

the trimming factor, ranging from 0 (no trimming) to 20, in steps of 2. The vertical axis represents the relative size of the data set after the trimming. In the figure, the largest reductions in the number of unique classes are achieved with trimming factors of 2 and 4, while the number of data samples does not follow the same trend. This can be seen as a noise-filtering step for the learning process.

With the final data set ready, the training of a machine-learned model can begin, as detailed in Section 6.3.

6.3 Training Models

In order to train a machine-learned model based on SVMs, two parameters must be defined: (a) the weighting used by the model to deal with (inevitable) misclassifications, and (b) the SVM kernel that will be used.

The misclassification cost provides a balance to learn a model that generalizes the training data well enough without overfitting, at the same time that it is flexible enough to deal with non-separable data caused by overlapping classes without underfitting. In either case, the resulting model will perform poorly. When selecting a kernel, there is a trade-off on the ability of the model to adapt to the correlations within the data and its capability to both train and perform predictions in a timely fashion.

The misclassification cost is discussed in Section 6.3.1, followed by the kernel selection in Section 6.3.2.

6.3.1 Cost Parameter for SVMs

The C cost parameter for SVMs defines the penalty of misclassifying a data instance when training a machine-learned model. Penalties too small may lead to a poor model, in the sense that the separating hyperplanes can be placed in a sloppy way. On the other hand, large misclassification costs can force the model to overfit to a specific training scenario which can be unrealistic. In this case, the model can also perform poorly.

It is normally the case that the ideal penalty is computed from cross-validation, *i.e.*, many models are created using different penalty values, so that a balanced cost can be inferred from the experimental results. The cross-validation is often performed with C ranging from 10^{-5} to 10^{5}, where C is enlarged by a factor of 10 on each iteration. The drawback is that when a test data set cannot be used to quickly evaluate the learned model, this process becomes prohibitive in terms of the time budget. In addition, larger values of C tend to extend the time for training a model since it becomes more difficult to find separating hyperplanes that do not violate the tighter constraints.

For example, when cross-validating C over 5 models, this process requires the training and evaluation of 55 models in order to identify the optimal C. If the data sets change, the process must be repeated because C is only optimal in the context of the data sets used for training. A common recommendation is to use $C = 1$, which does not artificially bias the learning process towards under or overfitting. In this study, the value $C = 10$ was experimentally determined to balance the quality of the learned model with practical training times. Values of C larger than 10 results in increasingly longer training times.

The next parameter when training an SVM, selecting a kernel function, is discussed in Section 6.3.2.

6.3.2 Kernel Selection

When selecting an SVM kernel to be used in a time-sensitive problem such as recommendations for a Just-in-Time compiler for Testarossa, there are two important factors to consider: (a) the dimensionality of the feature space, and (b) the time budgets, especially the time constraints when performing a prediction. These factors are tightly correlated.

The first factor considered is the dimensionality of the feature space. Lower dimensionalities usually benefit from non-linear kernels which map the original feature space into a higher-dimensional space. Conversely, high-dimensional spaces benefit less from the higher-dimensional mapping, which may not improve the learned model, but have an added computational cost. In some cases, non-linear kernels are too computationally intensive for high-dimensional problems, especially during the computation of a prediction.

In terms of performance, a linear kernel is very attractive due to the simpler nature of the computations performed. In some cases, as with LIBLINEAR, the implementation of the model is oriented to this specific kernel and tailored for larger data sets, giving better performance with equivalent accuracy to a regular SVM implementation [13].

It is often the case that there are special limitations on the time budget. Long training times reduce the possibility of investigating ideal parameters (*e.g.*: limiting the possible iterations for cross-validation), whereas long prediction times may hinder its applicability. A linear kernel, for example, is not guaranteed to train faster than a non-linear kernel, since a non-linear kernel is more flexible. On the other hand, a linear kernel such as LIBLINEAR delivers the fastest prediction-time responses regardless of the size of the training data set, making it very attractive for the time constraints found in Testarossa.

This study experimentally found that a non-linear kernel such as Radial Basis Function (RBF) (implemented by LIBSVM) has the lowest training times, but its prediction speed strongly depends on the size of the training input, and the resulting model can be very large. It was observed that LIBSVM trains a model in about 20% of the time used by LIBLINEAR for the same data set. However, prediction times for LIBSVM are much larger than LIBLINEAR. In some cases, a learned RBF model can take up to 660 ms to compute a prediction[1], which is usually many times longer than the time spent by Testarossa when compiling in the highest optimization level.[2] On the other hand, the same data set takes considerably longer to learn a model using LIBLINEAR, but the time to compute a prediction can be as low as 48 μs (4 orders of magnitude faster).

The next step, after training machine-learned models, is to deploy them with Testarossa so that it can perform method-specific compilations. The integration of such models with Testarossa is discussed in Chapter 7.

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

[2]In the experimental platform used in this study, Testarossa can compile a method in the highest optimization level in about 100 ms to 220 ms. The compilation time is highly dependent on the method being compiled, so these figures are illustrative.

Chapter 7

Integrating Compiler and Model

The learning-enabled Testarossa architecture is presented in Figure 7.1. In essence, its operation is similar to the data collection process. The initial decision to compile a method remains entirely up the the policies already in place in Testarossa (*a*). When such a 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 learning model (*d*). The details of this communication are discussed in Section 7.3. The model computes the classification for the features received, and sends back the result to the Strategy Control extension (*e*). As can be seen in the Figure, the arrow (*d*) is purposely thicker than (*e*), to illustrate that the outbound communication of the features is larger than the response (the encoded compilation plan modifier).

The Strategy Control extension installs the compilation plan modifier (*f*) in the same way as during data collection. In fact, most of the framework used for data collection is reused for the learning-enabled approach, with the components relevant only to data collection removed. With the compilation plan modifier in place, the optimizer is allowed to start. As happens during data collection, the optimizer is instructed to disable some of the transformations in the compilation plan that would be otherwise applied in its entirety (*g*). When the compilation finishes, 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 and, in the end, replacing the existing image of the method with the newly compiled version.

The communication steps and underlying pre-computation inside the machine-learned model is progressively described as follows. First, Section 7.1 discusses the normalization of the data prior to the model performing the prediction. Next the mapping from machine learning classes into compilation plan modifiers is discussed in Section 7.2. Finally, Section 7.3 consolidates all of the components to describe the communication protocol.

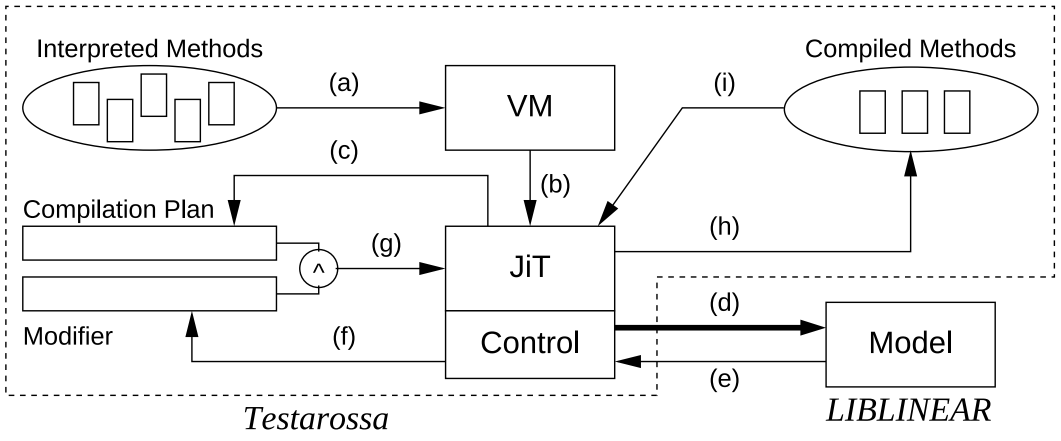

Figure 7.1: Architecture of the learning-enabled Testarossa. The modifier architecture is activated when the optimizer is ready to start. The strategy control extension computes the features of the method and sends them to the machine-learned model through a socket-based communication (*d*), that responds with a compilation plan modifier (*e*) that modifies the compilation strategy performed (*g*).

```
73
3    100000  353         0  0     15          0  0  1         0    0  ...
0    0       0.00636943  0  0.05  0.00157233  0  0  0.166667  0.1  0  ...
```

Figure 7.2: Format of a *scaling* file. The first line indicates the number of components in the feature vector, followed by the scaling parameters in a separate line each. The second line contains the shifting values that are subtracted from the feature vector, and the third line contains the scaling values used to multiply the shifted feature vector. Both sets of operations are performed in an element-wise fashion.

7.1 Normalization

When a training data set is generated, a set of normalization parameters are saved in a *scaling* file to be used by the model. Figure 7.2 illustrates the contents of a scaling file. The first line indicates how many components there are in the feature vector. The second and third lines contains the actual shifting and scaling parameters. The normalization is carried out using the linear transformation presented in Equation 7.1.

$$\vec{N} = \left(\vec{F} - \vec{O}\right) \circ \vec{S} \tag{7.1}$$

In the equation, $\vec{F}$ is the feature vector of the method being compiled, containing raw values as collected by Testarossa; $\vec{O}$ is the shifting vector; and $\vec{S}$ is the scaling vector. The operator $\circ$ denotes element-wise vector multiplication. The result is assigned to the normalized feature vector $\vec{N}$.

The shift-scaling operation uses the same parameters that produced the training dataset, so all training instances are normalized to the range $[0, 1]$. It is possible, however, that $\vec{N}$ contains values outside the range $[0, 1]$, but this is expected and the learned SVMs are able to

```
7275
4201   2438974982267015
5821   2622099574323670
5334   3623932176363988
2249   5531996405099509
5160   6257789759115178
623    8007126159813822
...
```

Figure 7.3: Format of an index file. The first line indicates how many mappings exist in the index file. Every following line is a pair in the form $\langle l, L\rangle$, where l is the class in the smaller space (required by LIBLINEAR/LIBSVM), and L is the actual encoding of the compilation plan modifier that is used in Testarossa.

handle this situation. The resulting $\vec{N}$ is then used by the model to predict the compilation plan modifier.

7.2 Class-to-modifier Mapping

The prediction on the normalized feature vector $\vec{N}$ produces an output class identifier G in the range $[1, 2^{31} - 1]$. Because this range is specific to the model implementation (both LIBSVM and LIBLINEAR use this range), and since it is not large enough to hold an encoded compilation plan modifier, the extensions to the model have to map it back to a meaningful compilation plan before responding to Testarossa.

The scheme in use is a lookup table, where each known class identifier is associated with a compilation plan modifier. This look-up table is prepared during the initialization of the model, and is loaded from an *index* file created during data set generation, as illustrated in Figure 7.3. The first line of the file indicates the number of entries in the lookup table, which is equal to the number of unique classes found during data set generation. Every following line contains a mapping from a class identifier (first column) to a compilation plan modifier (second column). The i-th element in the lookup table is then assigned the corresponding compilation plan.

When the model predicts G, the lookup table is consulted at the G-th index and the associated compilation plan modifier is sent back to Testarossa.

7.3 Socket-based Communication

The communication between Testarossa and the machine-learned model (illustrated in Figure 7.1 (d) and (e)) is carried out using *named pipes* [37], which in UNIX or UNIX-like environments are a form of Inter-process Communication (IPC) by means of a file-system-based pipe. A *named pipe* is a special file in the file system hierarchy in the sense that its contents are dynamically generated by another process, but it is attached to the file system.

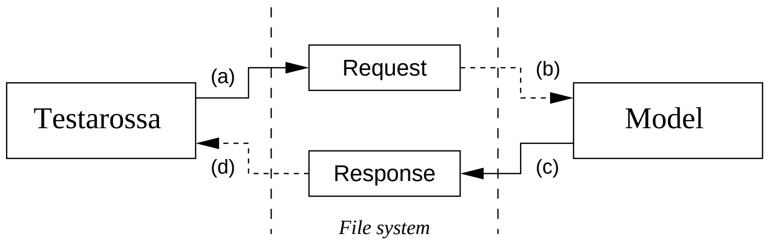

Figure 7.4: Socket-based communication between Testarossa and the model. ...

The data is transferred in a first-in first out (FIFO), producer-consumer fashion, with the producer opening the pipe for writing and the consumer opening the same pipe for reading. The consumer locks on reading the pipe until data is available; the producer usually does not lock on writes except when the in-kernel IPC buffer is full. Because pipes are byte streams, neither of the applications communicating can perform seeks, so they need to agree on a protocol for exchanging messages.

The communication is illustrated in Figure 7.4. A request is sent by Testarossa to the model by writing the feature vector $\vec{F}$ of the method being compiled to the model on the *requests pipe* (a). The message is composed of 71 32-bit unsigned integers, resulting in fixed-length messages of 284 bytes. All method features are thus converted from their native types, even though some of the features are smaller than 32 bits. This procedure avoids alignment issues[1] without adding complexity to either the sender or receiver at the cost of additional bytes transmitted.

The receiver attached to a machine-learned model (in this study, receivers have been incorporated into LIBSVM and LIBLINEAR) reads exactly 284 bytes from the *requests pipe* (b), and converts to the data format of the model. In both LIBSVM and LIBLINEAR cases, the unsigned integers are converted to double-precision floating-point, which is the base type used in those models. Before the model computes its prediction, the data received is normalized as detailed earlier in Section 7.1. When the prediction is ready, the output is converted to a compilation plan modifier as required by learning-enabled Testarossa (described earlier in Section 7.2), and written to the *responses pipe* (c). This write unblocks Testarossa, that was waiting for a response in the pipe (d).

The compilation plan modifier is installed and the optimizer is allowed to continue its execution.

[1]Most architectures require that the address of a memory data access be a multiple of its word size (often multiples of 4-byte), and thus aligned [2]. Unaligned accesses are possible but they incur in a significant overhead that must be either handled by the processor (if supported) or by software. Under some circumstances, an unaligned access can trigger a processor exception.

7.4 Trade-offs

The integration of learning-enabled Testarossa and machine-learned models by means of named pipes incurs an additional overhead when compared to the regular library- or linking-based approach. In the library-based approach, the machine-learned model is implemented as an external dynamic library that is loaded by Testarossa explicitly, whereas the linking-based approach embeds the compiled units of the model into Testarossa. There are, however, advantages in the communication approach used.

Complexity Embedding a machine-learned model implementation into Testarossa demands significant changes to the model implementation, so that it can properly make use of Testarossa's infrastructure (especially, the customized memory management mechanism in Testarossa).

Compatibility To integrate with Testarossa, the model must be implemented in a compatible programming language (C or C++), or there must exist language bindings that allow Testarossa to invoke the model.

Interchangeability As long as the implementation of the machine-learned model is properly augmented with the communication protocol, changing models does not require any change to the compiler, making it trivial to experiment with different models. If the model is embedded into Testarossa, changing models requires modifying and recompiling Testarossa, something that is not always possible (*e.g.*: a third-party that is not allowed access to Testarossa source code).

In practice, the overhead incurred by the extra communication is negligible, and often it is dominated by the computation cost of the model performing a prediction. The actual overhead depends on the platform. In the platform used for this study, the communication cost is approximately 230 µs per request.

The methodology used to evaluate the learned models is detailed in Chapter 8, including the performance results obtained.

Chapter 8

Experimental Evaluation

The learned models were deployed with Testarossa to evaluate whether they were able to capture patterns in the training data that lead to either (*a*) reductions in the compilation effort, and/or (*b*) improvements in the running time of the application.

The performance experiments were carried out in a multi-core cluster environment, which is detailed in Section 8.1. The method used to create the training data sets to perform leave-one-out cross-validation is detailed in Section 8.2, followed by the training times obtained when using LIBLINEAR, in Section 8.3. The methodology used to perform the performance experiments is detailed in Section 8.4. Next, the results obtained in the performance evaluation are discussed in Section 8.5. Finally, concluding remarks are presented in Section 8.6.

8.1 Experimental Setup

The platform used both for data collection and for tests consisted of a blade server with 16 nodes, 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. One of the nodes in the server acts as a management node, and was not used in any part of the experiments.

When performing data collection or a test, the respective node in the server had no other application competing for resources, besides the operating system and the regular set of system applications (for example, secure remote shell (SSH) and the system logger).

8.2 Model Training

In the experimental evaluation, five machine-learned models were trained with the data collected from the SPECjvm98 suite [34]. SPECjvm98, consists of the eight applications listed in Table 8.1. The first column of the table contains the benchmark name. The second column contains the abbreviated name for the benchmark, which is used in the tables and figures. The last column provides a brief description of the purpose of each benchmark

Table 8.1: Benchmarks included in the SPECjvm98 suite.

Benchmark Name	Short	Description
_201_compress	`co`	Compressor implementing the Lempel-Ziv method (LZW)
_209_db	`db`	Memory-based database application
_228_jack	`jk`	Java parser generator, an ancient version of JavaCC
_213_javac	`ja`	Java compiler from JDK 1.0.2
_202_jess	`je`	Java Expert Shell System, based on CLIPS expert shell system from NASA
_227_mtrt	`mt`	Threaded version of _205_raytrace, using two threads
_222_mpegaudio	`mp`	MPEG layer-3 audio decompressor
_205_raytrace	`rt`	Ray tracing application

Table 8.2: Benchmarks used for each training data set. The data sets used for training are uniquely identified, as indicated in the first column. The benchmarks that contributed with data collected for a given data set are marked with the symbol • in the respective column.

Data set	Collected application data merged				
	`co`	`db`	`mp`	`mt`	`rt`
1	•	•	•	•	
2	•	•	•		•
3	•	•		•	•
4	•		•	•	•
5		•	•	•	•

application. The SPECjvm98 suite actually includes a nineth application, `_200_check`, that is only used to determine if the JVM at hand is capable of executing the benchmark suite. Therefore, `_200_check` is not used for either data collection or model testing.

The data-collection process was not successful on all benchmark applications. Data collection for `_213_javac` and `_228_jack` generated compilation inconsistencies in a large number of trials. On the other hand, `_202_jess` succeeded for the randomized search, but faced instrumentation errors when using the progressive search. To ensure fairness in the evaluation, `_202_jess` was not used to create training data sets.

The data sets created using the data collected from the five successful SPECjvm98 benchmarks are listed in Table 8.2. The first column in the table contains a unique identifier assigned to each training data set. The remaining columns indicate whether data from a given benchmark was included or not in the respective data set. If the column contains the symbol •, data from the benchmark was used in the respective data set. For example, data set 1 used data from benchmarks `co` (compress), `db`, `mp` (mpegaudio), and `mt` (mtrt).

Each training data set merges data collected from five distinct benchmarks, leaving the fifth one aside. This way, the resulting models can be evaluated using leave-one-out cross-validation, where one of the applications will not contribute to the training data set so that the data is not used in computing that model.

Data sets are created using data collected using different searching approaches. Data sets prefixed with the letter **R** use data collected using the randomized search (*e.g.*: R2 contains data from benchmarks `co`, `db`, `mp`, and `rt`). The data sets prefixed with **P**, in turn, use data collected using the progressive search. A third set of models, prefixed with the letter **H**, uses data from both approaches. For example, data set H2 contains data from `co`, `db`, `mp`, and `rt` benchmarks, for both randomized search and progressive search approaches. In total, there are 15 data sets, 5 for each exploration approach alone and another 5 where their data is merged.

The data collected for the benchmarks is detailed in Table 8.3. The table is divided into three groups, where each group refers to a different approach used to generate compilation-plan modifiers. Across all three groups in the table, the first column contains the name of the benchmark application used to collect data. The second column contains the number of data instances produced. Each data instance describes a compilation experiment performed. The third column contains the number of unique classes in the set of data instances. A class is a compilation-plan modifier that was mapped into a smaller integer range ($[1, 2^{31} - 1]$) as a requirement for the machine-learned model implementation, LIBLINEAR. The fourth column contains the number of unique feature vectors in the set of data instances. As far as the model is concerned, unique feature vectors refer to distinct methods. However, because feature vectors summarize a method using a limited number of features, distinct methods can still be treated as one if their feature vectors are identical. The last column in the table contains the class-to-instances ratio, computed from the data collected (*i.e.*, the average number of instances for each unique class in the data).

The third group in Table 8.3 refers to training data produced by merging both approaches used to generate compilation-plan modifiers, thus called *hybrid*. Both the number of data instances and unique classes are, roughly, twice as large, while the number of unique feature vectors is only slightly larger. The reason for such a small increase is because the set of methods compiled across multiple JVM executions tends to be similar. However, because Testarossa uses dynamic information from the application executing to decide which methods to compile, the set of methods compiled is different across distinct runs.

The parameters used to generate the data sets were progressively tuned during the course of many data set generations. In this scenario, a maximum of 3 compilation-plan modifiers were selected during the ranking process, for each unique feature vector. In addition to the maximum of 3 modifiers, the Top-$M\%$ cut-off value was set to 5%. This cut-off of 5% restricts the modifiers selected to those having at least 95% of the ranking value of the best modifier for a given feature vector. For the learning process, selecting up to 3 modifiers allows the model to better aggregate neighboring classes in the feature space, while the

Table 8.3: Characterization of the training data. The benchmark applications are used during data collection, with the two distinct approaches to generate compilation-plan modifiers. The size of the training data is given by the number of data instances. The modifiers (classes) are reused in different compilations so the number of unique classes is smaller than the number of data instances. A distinct set of methods are compiled for each benchmark, and the number is approximated in the column containing the number of unique feature vectors. The last column contains the ratio of data instances for each unique class in the data set.

Progressive Randomized Search				
Benchmark	**Data Instances**	**Unique Classes**	**Unique Feature Vectors**	**Class-to-instances Ratio**
`compress`	76,109	14,732	508	1:5.17
`db`	51,118	12,475	499	1:4.10
`mpegaudio`	250,951	24,456	760	1:10.26
`mtrt`	121,461	14,457	828	1:8.40
`raytrace`	116,159	14,768	922	1:7.87

Randomized Search				
Benchmark	**Data Instances**	**Unique Classes**	**Unique Feature Vectors**	**Class-to-instances Ratio**
`compress`	111,046	14,587	508	1:7.61
`db`	147,892	14,617	485	1:10.12
`mpegaudio`	278,838	11,134	662	1:25.04
`mtrt`	172,541	11,154	694	1:15.47
`raytrace`	152,296	11,135	654	1:13.68

Hybrid: Randomized and Progressive Randomized Search				
Benchmark	**Data Instances**	**Unique Classes**	**Unique Feature Vectors**	**Class-to-instances Ratio**
`compress`	187,155	29,319	628	1:6.38
`db`	199,010	27,092	611	1:7.35
`mpegaudio`	529,789	35,590	867	1:14.89
`mtrt`	294,002	25,611	973	1:11.48
`raytrace`	268,455	25,903	1,022	1:10.36

5% cut-off removes modifiers that underperform the best modifier. The hypothesis is that the model will be able to achieve a better agreement on densely-populated regions of the feature space. In this case, it is possible that a modifier located on the centroid of such a dense region in the feature space might becomes a support vector when, in fact, a better compromise is to select a more representative class in that region for a support vector.

The final data sets are detailed in Table 8.4. Each row in the table presents information for a final data set, identified, in the first column of the table. The table is divided into three groups, one for each approach used to generate compilation-plan modifiers. In turn, each of the groups is divided into collected data and ranked data. The collected data refers to the data collected from the benchmarks and merged into a intermediate data set. When the intermediate data set is processed with the ranking tools, a final data set consisting of ranked data is produced. The final data sets are then used to learn the respective machine-learned models.

The column *Data Instances*, across all three groups of data sets, contains the number of data instances after merging the data collected from the benchmarks, that are used in the respective data set. The column *Unique Classes* contains the number of unique classes in the merged data. A class is a compilation-plan modifier mapped into the range accepted by the model implementation. The column *Unique Feature Vectors* approximates the number of unique methods compiled during data collection. The next column, *Ratio*, presents the average ratio of data instances for each unique class in the data, computed for every unique class in the data set.

Across all groups of data sets in the table, the columns under *Ranked Data* present the information analogous to the *Collected Data*. The size of the final data set is presented in the column *Training Instances*. The column *Training Classes* contains the resulting number of unique training classes for the respective data set. Next, the column *Training Feature Vectors* presents the number of unique feature vectors in the ranked data. Finally, the column *Ratio* contains the ratio of training instances for each unique training class.

The first ranking parameter, limiting the number of selected compilation-plan modifiers to a maximum of 3 for each unique feature vector, implies an upper bound limit on the ratio of 1:3. However, the second ranking parameter selects only those modifiers that have at least 95% of the ranking value of the best modifier for each unique feature vector. This cut-off makes the effective ratio smaller than the upper bound, if the modifiers used in the data-collection compilations have a ranking value smaller than 95% of the best modifier, for each unique feature vector.

The optional operation on a training data set, trimming, is only necessary if the data set is too large. For the data sets described in Table 8.4, this is not the case, thus no trimming was required. The maximum data set size depends on the available resources for training. In

Table 8.4: Characterization of the final data sets. Each row in the table represents a data set, identified by a letter and a number. The letter indicates the approach used to generate compilation-plan modifiers. The number indicates which of the benchmark applications contributed with training data. The size of the data sets are reported in the *Data Instances* and *Training Instances* columns, before and after the ranking process, respectively. The number of unique classes, which encode modifiers, is displayed in the same form, under *Unique Classes* and *Training Classes* columns. The number of unique feature vectors is displayed in the following columns, *Unique Feature Vectors* and *Training Feature Vectors*. Finally, both *Ratio* both report the ratio of instances for each unique class, before and after the ranking process, respectively.

Data Sets using Progressive Randomized Search								
	Collected Data				**Ranked Data**			
Data Set	**Data Instances**	**Unique Classes**	**Unique Feature Vectors**	**Ratio**	**Training Instances**	**Training Classes**	**Training Feature Vectors**	**Ratio**
P1	499,639	63,657	1,412	1:7.87	2,065	1,074	1,412	1:1.92
P2	494,337	63,960	1,513	1:7.85	2,184	1,101	1,513	1:1.98
P3	364,847	54,009	1,250	1:7.73	1,752	854	1,250	1:2.05
P4	564,680	65,897	1,480	1:6.75	2,135	1,101	1,480	1:1.94
P5	539,689	63,762	1,492	1:8.57	2,160	1,108	1,492	1:1.95

Data Sets using Randomized Search								
	Collected Data				**Ranked Data**			
Data Set	**Data Instances**	**Unique Classes**	**Unique Feature Vectors**	**Ratio**	**Training Instances**	**Training Classes**	**Training Feature Vectors**	**Ratio**
R1	710,317	51,489	1,200	1:13.80	2,323	1,428	1,200	1:1.63
R2	690,072	51,470	1,178	1:13.41	2,302	1,380	1,178	1:1.67
R3	583,775	51,490	1,000	1:11.34	1,884	1,164	1,000	1:1.62
R4	714,721	48,007	1,155	1:14.89	2,254	1,400	1,155	1:1.61
R5	751,567	48,037	1,148	1:15.65	2,224	1,413	1,148	1:1.57

Data Sets using both Randomized and Progressive Randomized Search (Hybrid)								
	Collected Data				**Ranked Data**			
Data Set	**Data Instances**	**Unique Classes**	**Unique Feature Vectors**	**Ratio**	**Training Instances**	**Training Classes**	**Training Feature Vectors**	**Ratio**
H1	1,209,956	115,146	1,592	1:10.51	2,712	1,649	1,592	1:1.64
H2	1,184,409	115,430	1,651	1:10.26	2,780	1,628	1,651	1:1.71
H3	948,622	105,499	1,395	1:8.99	2,295	1,363	1,395	1:1.68
H4	1,279,401	113,904	1,625	1:11.23	2,733	1,635	1,625	1:1.67
H5	1,291,256	111,799	1,618	1:11.55	2,720	1,641	1,618	1:1.66

Model	Training Times for Different Misclassification Costs					
	C = 1		C = 10		C = 100	
	Time	Iterations	Time	Iterations	Time	Iterations
P1	23"	32	1'19"	359	5'17"	3,330
P2	27"	53	1'25"	313	6'21"	3,382
P3	15"	33	53"	409	2'52"	2,984
P4	24"	35	1'21"	336	5'02"	2,964
P5	28"	41	1'24"	396	5'28"	3,180
R1	49"	39	3'24"	410	15'56"	5,239
R2	50"	42	3'25"	524	15'01"	4,291
R3	33"	43	2'24"	557	8'55"	4,625
R4	49"	46	3'07"	396	16'12"	4,646
R5	49"	39	2'55"	403	13'24"	3,922
H1	1'07"	58	4'39"	784	17'38"	6,644
H2	1'12"	76	3'51"	541	18'37"	6,539
H3	47"	64	3'01"	667	11'42"	6,540
H4	1'03"	63	3'36"	526	16'02"	5,664
H5	1'02"	55	4'32"	613	19'39"	7,975

Figure 8.1: Model training times using LIBLINEAR. The training times reported are in minutes and seconds. Multiple models are trained, varying the misclassification cost from $C = 1$ to $C = 100$. The number of iterations required by the model to optimize the quadratic optimization is also presented.

the experimental platform used, data sets with $80,000$ data instances or larger demand more resources than those available, causing the training application to become I/O-dominated due to excessive paging of the operating system.

The final data sets are then used to train machine-learned models using LIBLINEAR. The details are discusses in Section 8.3.

8.3 LIBLINEAR Training

With the final data sets ready, multi-class machine-learned models are trained using LIBLINEAR [13]. The classification model used is the multi-class classifier using the SDM [24] algorithm. Previous experiments revealed that, for this study, misclassification costs of $C = 0.1$ or less produce poor models. Thus, models are trained only for $C = 1$ and higher misclassification costs. All other training parameters used are the default ones in LIBLINEAR.

Table 8.1 illustrates the training times obtained with LIBLINEAR for each model. The column *Model* identifies all machine-learned models trained, with the letter indicating the approach used to generate compilation-plan modifiers, and the number indicating the set of SPECjvm98 benchmarks that contributed with data collected. Multiple models are generated from each training data set, with a misclassification cost varying from $C = 1$ to $C = 100$. As the misclassification cost is increased, the training process takes longer because it requires additional iterations to find the maximum separating hyperplanes.

The training process was tuned over several months to ensure timely learning of models. The set of features collected has the largest influence in the training times, thus being the focus of the tuning process. A larger set of features requires more computation by the training application, and more resources. In addition, a larger set of features can, unwillingly, produce distinct feature vectors for the same method. The effects are twofold: (*i*) the ranking process cannot operate optimally since the same method is represented with multiple feature vectors; and (*ii*) the training process is significantly extended, because the training data sets are larger.

In previous attempts, the resulting data sets used for training were much larger, requiring the trimming process (detailed in Section 6.2). The trimming kept data sets under $80,000$ training instances, which is the practical limit for training data sets in the setup used. Such training data sets resulted in training times spanning from 2 to 35 days, using $C = 1$. Worse, training times with misclassification costs higher than $C = 1$ were not feasible for most of the data sets.

The initial set of features, which included 118 numerical features, was reworked over the course of many iterations, in collaboration with IBM engineers working on Testarossa. While some machine-learning algorithms are able to discover useful features on their own[1], current SVM implementations are not able to do so.

The learned models are then used in conjunction with Testarossa to evaluate their effectiveness. The evaluation approach is described in Section 8.4.

8.4 Experimental Methodology

In order to verify the prediction performance of the learned models, the SPECjvm98 benchmarks were executed in a batch session, for each of the learned models, using the largest inputs for each benchmark. The benchmarks were executed 30 times in succession, and the total running time was recorded. The repeated execution is important to account for both internal (*e.g.*: garbage collection and set of methods compiled) and external factors (*e.g.*: thread scheduling by the operating system and I/O latencies). In addition, the internal iteration count of each benchmark is set to 10, meaning that the core of the benchmark will be repeated 10 times, to dilute the startup effects. The main results consider the time to complete all 10 iterations of each benchmark measured.

[1]*Decision Trees* [19], for example, compute the amount of information that each feature adds to the model, producing a hierarchical decision tree from the most informative feature to the least informative. The resulting tree is usually pruned to remove less-informative features below a defined threshold, thus performing feature selection naturally.

Both startup performance and long-term execution performance are important. Startup performance relates to the ability of the JVM to make application progress quickly. In this case, the tradeoff in compilation time and performance of the code generated is more critical: the more the JVM invests in the compilation, the later the application makes progress (*i.e.*, executes). If the application is short-running, the compilation costs can hinder a significant portion of the benefits.

On the other hand, long-term performance refers to methods that account for a large portion of the running time of the application. In the `compress` benchmark, for example, the method `Compressor.compress()V` accounts for a large portion of the execution. The tradeoff, in this case, is different than for startup performance: all code transformations that improve the execution performance of the method are likely to pay off by reducing the execution time of the future invocations of the method.

Both the data-collection process and the evaluation use only the *hot* optimization level. This is because the learned SVM is not able to distinguish between optimization levels, requiring multiple SVMs, one for each optimization level. Such a multi-level approach is necessary because compilation-plan modifiers are ranked considering a specific optimization level. While the modifier itself is neutral to the optimization level, using a modifier ranked for a different optimization level is inconsistent, and the outcome is unknown.

The experimental results are discussed in Section 8.5.

8.5 Experimental Results

The results reported are relative to those obtained with Testarossa using only the *hot* optimization level. The results also include the performance of Testarossa operating under adaptive mode, which is the default mode of operation. The first approach discusses is the progressive randomized search, in Section 8.5.1.

8.5.1 Progressive Randomized Search Models

The performance results obtained with the models trained using data collected with the progressive randomized search are presented in Figure 8.2. For each set of benchmarks, the leftmost bar is the 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 using the progressive search approach to generate compilation-plan modifiers.

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 fourth models

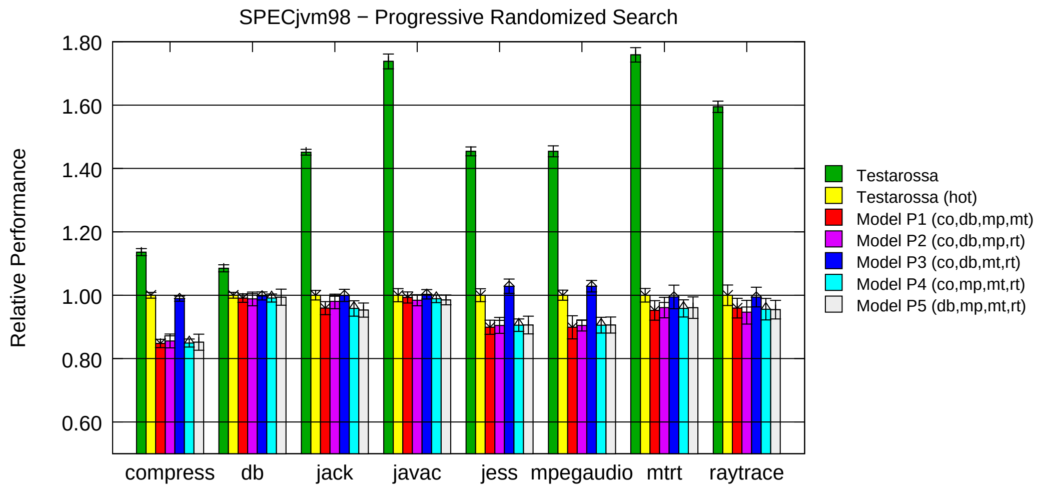

Figure 8.2: Performance results using progressive randomized models. The performance is relative to Testarossa operating only at the *hot* optimization level (second bar for each benchmark).

($X4$, where X is the approach used to generate modifiers) 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.

For five of the benchmarks, the models using progressive randomized search can obtain equivalent performance as that obtained by Testarossa using only the *hot* optimization level. For the other three, `compress`, `jess`, and `mpegaudio`, the learned models actually hurt the performance. The exception is model P3, which is able to match the performance of Testarossa *hot* for all benchmarks.

The performance for `compress` is the most affected across all benchmarks. The slowdown can be better understood from the per-iteration samplings, as shown in Figure A.1 on page 91. In the very first iteration of the benchmark, all models but P3 start with longer execution time. Because the code generated executes more slowly than the code generated by Testarossa *hot*, the performance difference grows larger at each benchmark iteration. The same behavior can be seen when inspecting the per-iteration execution time for `jess`, in Figure A.13 on page 97.

The case for `mpegaudio`, in Figure A.16 on page 98, is also similar—except for model P3. Model P3 starts slower than Testarossa under adaptive operation, but the graph suggests that the compiler using model P3 has a chance of reversing the difference. To determine whether this is the case, a similar experiment was conducted, but this time using 50 internal benchmark iterations instead of 10, and measuring only Testarossa under adaptive mode, Testarossa *hot*, and Testarossa using model P3. Figure 8.3 presents the results obtained.

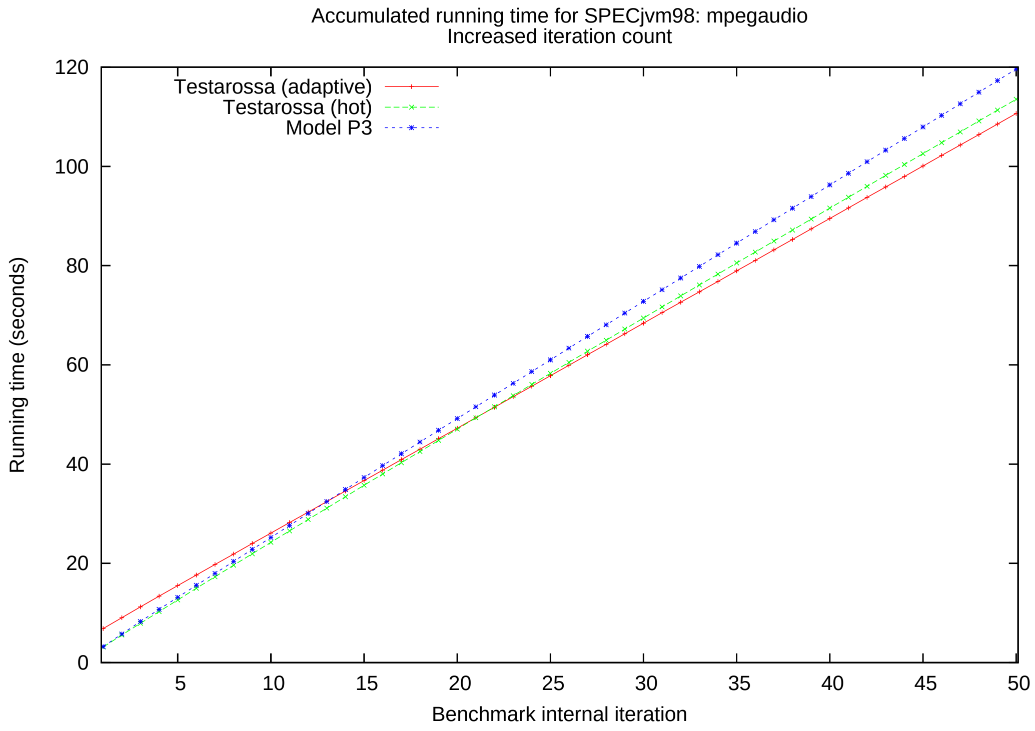

Figure 8.3: Additional experiment on `mpegaudio` using model P3. The difference in this experiment is the number of internal benchmark iterations, increased to 50. The graph plots the accumulated execution time until i-th iteration, averaged across 30 executions of the benchmark for each compiler.

Each line in the graph corresponds to a compiler and shows the accumulated running (on average) for each of the 50 iterations. In the modified experiment, Testarossa under adaptive operation starts slower than both Testarossa *hot* and Testarossa using model P3. In this case, Testarossa *hot* is always faster than the model. Adaptive Testarossa is able to outperform Testarossa using the model at the 13-th benchmark iteration, and at the 20-th iteration it outperforms Testarossa *hot*.

Figure 8.4 presents the compilation times, relative to Testarossa using only the *hot* optimization level. Because the compilation plans resulting from the models using progressive randomized search are very similar to the original plans included in Testarossa, the compilation times are very similar to Testarossa *hot*. In the cases where the difference is statistically significant, for all progressive models expect P3 on `jess` benchmark, the number of compilations performed is slightly larger. While Testarossa *hot* performed 366 compilations (on average), the progressive models performed 378 (on average), except for P3, with 367 compilations. When using the models, the compiler performs less code transformations on average (since some of the transformations are disabled), and thus is able to perform additional compilations.

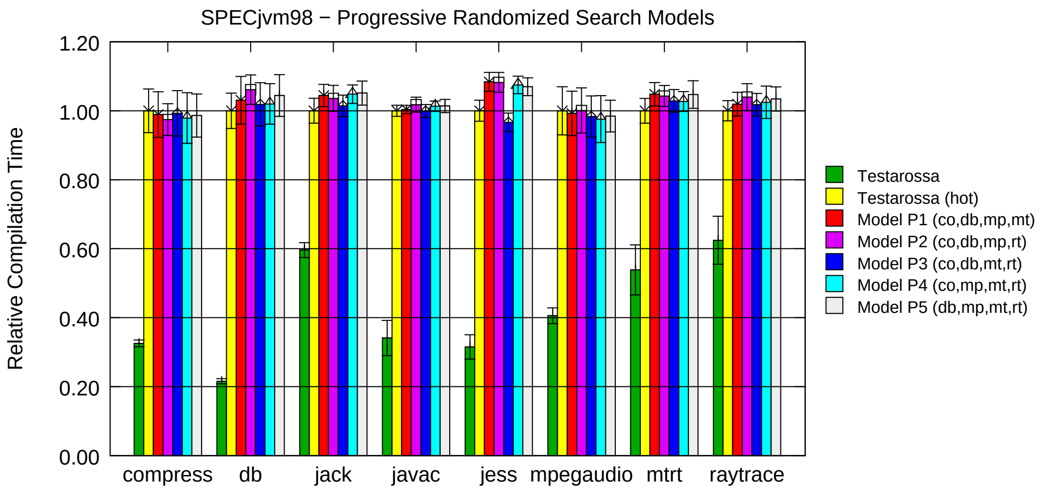

Figure 8.4: Compilation time using progressive randomized search models, relative to Testarossa *hot*. Because the resulting compilation plans are similar to the original plans included in Testarossa, the compilation times when using the models are very similar to the compilation times of Testarossa *hot*.

Section 8.5.2 discusses the results obtained with the models using randomized search.

8.5.2 Randomized Search Models

Figure 8.5 presents the performance results obtained when using models trained with data collected using the randomized search approach. The layout of the figure is the same as for the progressive randomized search models. For most applications (`compress`, `db`, `jess`, `mpegaudio`, `mtrt`), the models using randomized search are not able to outperform Testarossa, either under adaptive operation or using only the *hot* optimization level. However, for `jack` and `raytrace`, different models are able to reach the performance of Testarossa *hot* (R2 and R3, respectively).

For `javac`, all models are able to outperform Testarossa *hot* by a significant factor, approaching the performance obtained by Testarossa under adaptive operation. This case can be better understood by inspecting the accumulated running time per-iteration, in Figure A.11 on page 95. The compiler using the models start with a smaller running time, but Testarossa is able to reverse the difference and complete the 10-th benchmark iteration with a smaller accumulated running time.

Figure 8.6 presents the compilation times relative to Testarossa *hot*. Because the randomized search generates compilation-plan modifiers that disable a larger number of code transformations, the compilation times are significantly reduced. The compilation times when using the models are smaller than those obtained by Testarossa under adaptive operation, and significantly smaller than Testarossa *hot*. In addition, the number of compilations performed when using the models is 2% to 11% larger.

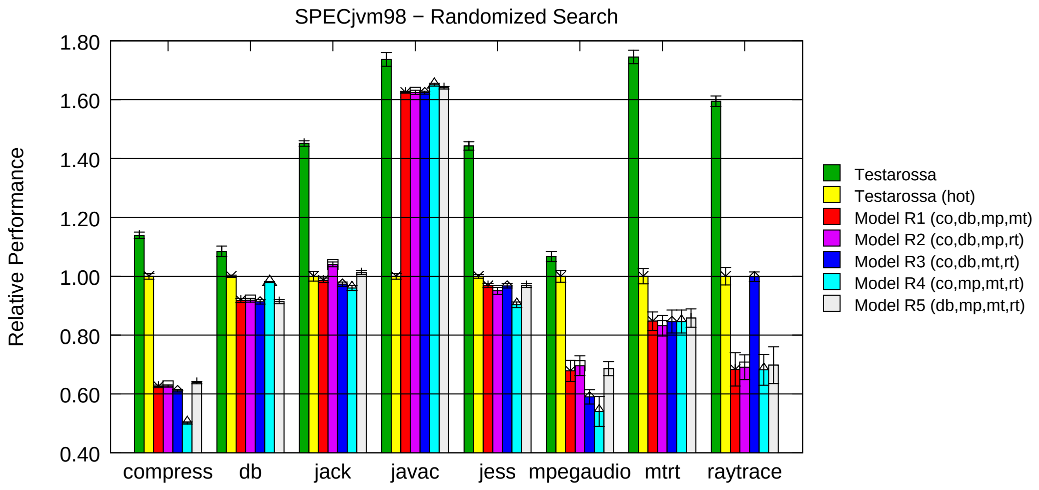

Figure 8.5: Performance results using randomized models. The performance is relative to Testarossa operating only at the *hot* optimization level (second bar for each benchmark).

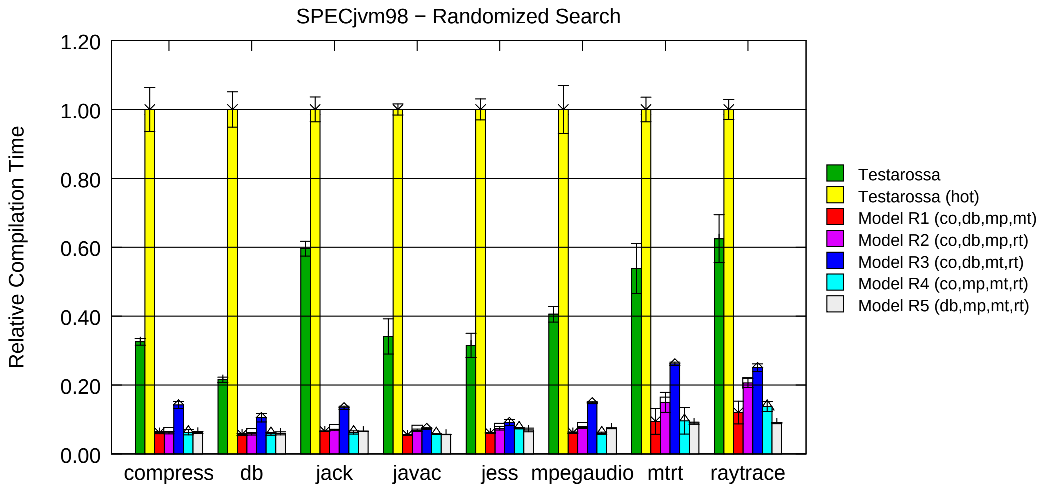

Figure 8.6: Compilation time using randomized search models, relative to Testarossa *hot*. The resulting compilation plans when using the randomized models are significantly different, and the compilation times are considerably smaller.

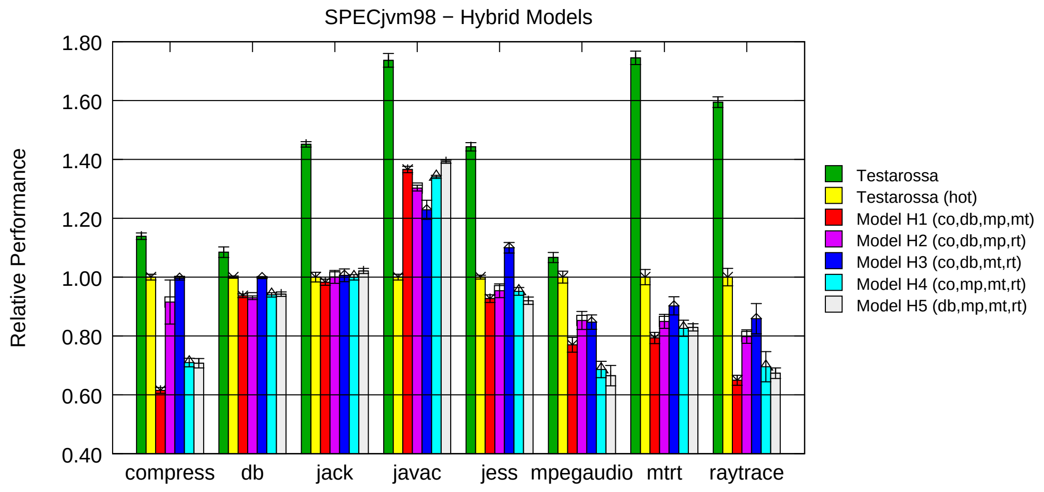

Figure 8.7: Performance results using hybrid models. The performance is relative to Testarossa operating only at the *hot* optimization level (second bar for each benchmark).

Section 8.5.3 presents the results obtained when using the hybrid models, using the data from both approaches that was used to generate compilation-plan modifiers.

8.5.3 Hybrid Models

The performance results obtained with the models using the hybrid approach, that mixes data from both the progressive randomized search and randomized search approaches, is presented in Figure 8.7. For most applications, `compress`, `db`, `jack`, `javac`, and `jess`, at least one of the learned models is able to reach the performance of Testarossa *hot*. In the case of `jess`, model H3 is able to outperform Testarossa *hot*. A similar behavior with `javac`, when using the randomized search models, repeats itself with the hybrid models, but with smaller significance. As detailed in Figure A.12 on page 96, the *hot* optimization level is not appropriate for the methods compiled in `javac` since the compilation effort outweighs the benefits. The learned models were able to tailor the optimization level on a per-method basis, but were not able to outperform Testarossa under adaptive operation.

Figure 8.8 presents the compilation times relative to Testarossa *hot*. For all benchmarks, the compilation times when using the models are smaller than those obtained with Testarossa *hot*. Because the hybrid models merge the data from the two approaches implemented to generate compilation-plan modifiers, the resulting compilation time is not as high as that obtained with progressive randomized search, but also not as low as when using the randomized search. Across `jack`, `javac`, `mpegaudio`, `mtrt`, and `raytrace` benchmarks, the compilation times, for at least one of the models, is smaller than Testarossa under adaptive operation. In addition, the number of compilations performed when using the models is always larger (from 1% to 11%) than both Testarossa under adaptive operation

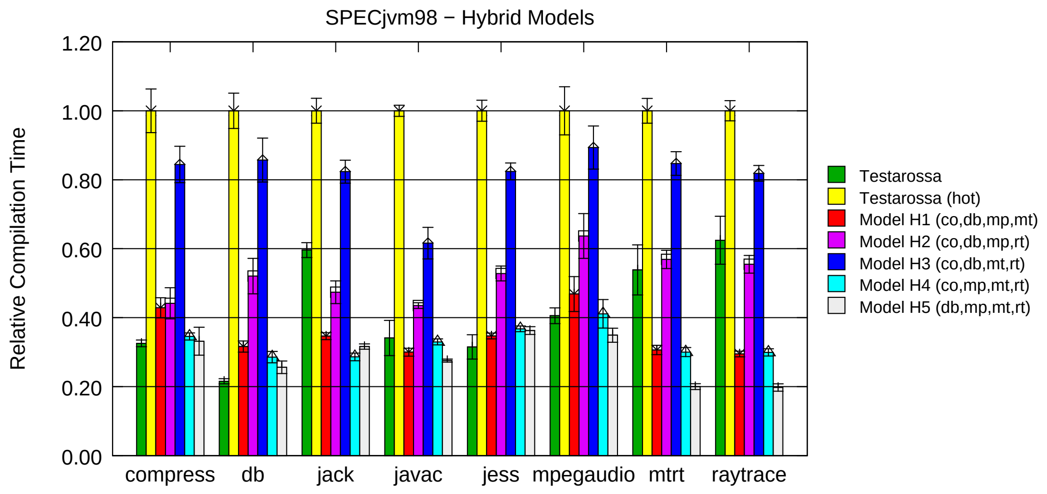

Figure 8.8: Compilation time using hybrid models, relative to Testarossa *hot*.

and Testarossa *hot*.

Section 8.6 presents final considerations for the chapter.

8.6 Concluding Remarks

The training process of machine-learned models used in this study, after many iterations, is able to generate learned models in a few minutes. The set of features currently in place allows the ranking process to generate small data sets from a rather large input of collected data. The overhead of using such models, in the platform used for the experiments, is about 230 µs per compilation. The total overhead depends on the number of methods compiled, but frequently accounts for less than 0.2% of the execution time of the application.

In the experiments performed, Testarossa, under its normal mode of operation (adaptive, multi-level optimization), consistently outperforms all other versions of the compiler (including Testarossa using only the *hot* optimization level, the second highest), 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. In order to achieve the same performance as Testarossa under adaptive mode, multiple machine-learned models must be trained, one for each optimization level. The data-collection process must explore alternative compilation-plan modifiers considering this space enlarged by multiple optimization levels.

The inclusion of machine-learned models into compilers, while not yet common, is not new. Chapter 9 discusses related work done by other authors.

Chapter 9

Related Work

Traditionally, the effectiveness of optimizing compilers is largely due to many person-years of hand-tuning. Most optimizing compilers have numerous internal parameters, such as loop-unrolling factors and inlining thresholds, that are set to values that lead to better code in numerous scenarios. Tuning these parameters is a highly complex task because many code transformations affect each other in non-obvious ways. In addition, it is not uncommon that parameters leading to improvements in a set of programs produce poor results in another set.

The relevancy of this issue has led to extensive research in order to improve optimization settings in compilers. The techniques proposed in such research have three key attributes: (a) compilation mode, (b) compilation granularity, and (c) approach used to explore compilation parameters.

The *compilation mode* can be either static, when code is generated in advance, or dynamic, when code is generated (or modified) while the application is running. For example, JiT compilers employ dynamic compilation. The goal of a static compiler is to spend as much time as is profitable during the optimization stage because, normally, the code generated does not change during execution. On the other hand, dynamic compilers have a limited compilation budget and require a balance between code quality and compilation effort. The *compilation granularity* refers to the scope of the proposed technique. Fine-grained techniques operate at the method (or procedure) level, whereas coarse-grained techniques consider the program in chunks (compilation units), as a whole, or even a set of programs. The extensions to Testarossa described in this thesis operate at the method level. Finally, the *exploration approach* relates to the core methods used in order to improve optimization settings. In the context of works related to this thesis, such methods are either based on machine-learned models or search techniques.

Some research focuses solely on a small set of compilation heuristics because of the complexity of creating, tuning, and maintaining those heuristics. This type of research is discussed in Section 9.1. On the other hand, there is a large body of work focusing on multiple code transformations and their interrelation. Works concerning numerous code transformations, or the complete set of transformations available in a compiler, are discussed in Section 9.2.

9.1 Individual Heuristics

One common optimization technique for exploiting parallelism at the machine-instruction level is *instruction scheduling*, on pipelined architectures. Pipelined architectures decompose the execution of instructions into micro-operations, which can be efficiently implemented in hardware—especially when operating at high clock frequencies. With the knowledge about the target architecture, the compiler can issue instructions with higher latencies earlier in a trace of code (*e.g.*: multiplication and division instructions), so that by the time the result of such an instruction is needed, the instruction has completed its execution and the result is available. While instruction scheduling is an NP-hard problem, an optimal solution is feasible when considering small units of code, such as *basic blocks*.

Moss *et al.* evaluate the use of supervised machine-learned models to perform instruction scheduling using a small set of features obtained from basic blocks [31]. The models are trained with both positive and negative examples of schedulings. In the evaluation, the models are compared to a hand-tuned instruction scheduler from the chip manufacturer and also with available production compilers. The learned models performed well enough to outperform the production compilers, but were not able to outperform the hand-tuned instruction scheduler. This work was later extended [8] to the Jikes Research Virtual Machine (RVM) [38], a Java JiT compiler. A model was trained in order to predict whether a basic block would benefit from instruction scheduling. Not all basic blocks benefit from instruction scheduling and instruction scheduling is a time-consuming code transformation. Therefore, the learned model was able to reduce the compilation effort while retaining most of the benefits from instruction scheduling on the remaining blocks. These findings are consistent with this thesis. For some applications, compilation-plan modifiers generated with the randomized search approach are able to reduce compilation time significantly, with low or no run-time degradation.

Stephenson *et al.*present a methodology for tuning multiple individual heuristics in a compiler [36]. Models using genetic programming start with base heuristics, with the goal of finding attributes that (*i*) describe the programs being compiled and (*ii*) are expressive with regard to the heuristic being tuned. The attributes are expressions derived from the intermediate representation used by the compiler, and the feedback to the algorithm is the difference in running time of the program. The algorithm evolves parameters for the set of programs, but is also able to generalize these parameters, to some extent, for new (unseen) instances of programs.

In a different work, Stephenson *et al.* describe the use of supervised machine-learned models used to learn loop-unrolling factors [35]. The authors use two distinct machine-learning algorithms: SVMs and Nearest Neighbors (NNs).[1] Loop unrolling is a code optimization technique where the body of the loop is replicated a few times by the compiler, in order to reduce the overhead caused by the code that checks the loop condition. As the authors report, the ideal loop-unrolling factor depends on the loop, and the models are trained to predict the best unrolling factor on a per-loop basis. The learned models lead to modest, but consistent, improvements in the running time of applications.

In an earlier attempt by Monsifrot *et al.*, a loop-unrolling heuristic was built based on a machine-learned model [30]. Loops are described by a handful of features obtained from the intermediate representation of the code, associated with a running time value compared to the original heuristics in the compiler. The model is trained with both positive and negative examples, and the compiler is modified to use predictions made by the model. Besides achieving improved performance when compared to the baseline compiler, the heuristics generated are architecture-specific, *i.e.*, the machine-learned model can be used to automatically generate heuristics appropriate to the platforms supported by the compiler.

In the approach described in this thesis, none of the internal heuristics from Testarossa are changed, with the exception of the compilation plan, that is tailored for each compiled method. Thus, whenever a code transformation is applied by Testarossa (for example, any of the many loop transformations available in Testarossa), it is applied using unmodified heuristics for the transformation, if any.

Section 9.2 discusses research concerning a larger (or complete) set of code transformations in a compiler.

9.2 Compiler Optimization Flags

The search for better compiler optimization flags gave rise to a wide range of approaches. These approaches can be organized into (*a*) iterative compilation, (*b*) automatic tuning of compiler heuristics, and (*c*) fine-grained compilation. The first approach, iterative compilation, is discussed in Section 9.2.1.

9.2.1 Iterative Compilation

Iterative compilation is the process where a single program is continuously compiled with different optimization flags, until a stopping criteria is met. The criteria can either be a speedup factor in comparison to a baseline version of the program (*i.e.*, with a reduced set of code optimizations applied), or a compilation budget (a deadline to release the application).

[1]For the models using Nearest Neighbors algorithm, the radius was determined experimentally from the data.

Iterative compilation is more useful for smaller applications (*e.g.*: for embedded environments), as well as those with long release cycles. Because iterative compilation can be very time-consuming, most researchers focus on speeding up the search for ideal compilation flags for the program.

Agakov *et al.*describe a methodology to accelerate iterative compilation using machine-learned models [3]. The model is learned with a set of training programs to explore the optimization space. The learning instances consist of ordered sets of code transformations. When an unseen program is submitted to the compiler, the exploration is focused on areas of the optimization space that are known by the model to yield improvements. The authors report significant speedups at the second iteration of the compilation. While it is possible to change the order of code transformations, or even derive a distinct compilation plan, doing so greatly enlarges the optimization space to search. This enlargement is due to the non-additivity of code transformations, *i.e.*, the order in which code transformations are applied affects the code generated. Thus, the original problem of predicting a method-specific compilation plan becomes, in addition, a phase-ordering problem.[25]

Fursin *et al.* describe the framework that led to MILEPOST GCC. They initially focused on evaluating more permutations in the optimization space [15]. Later they described a complete iterative compilation framework, capable of self-adjusting on different platforms [16]. 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. The implementation is currently able to control all aspects of the underlying compiler, GCC, and is able to replace the original heuristics altogether.

Besides relying on an iterative compilation approach, there two similarities to the approach described in this thesis. First, because the optimization space is large (GCC also implements a large number of code transformations), the exploration approach does not consider the order in which code transformations are applied, but only whether a transformation is or is not applied. Second, the machine-learned model is deployed externally to the compiler, as a Matlab server, responding to compilation requests. One key difference is that the machine-learned models are created on a per-application basis, *i.e.*, they are trained in the context of an individual application.

Cavazos *et al.* [7] describe the methodology used to train a program-specific machine-learned model. The feature space is composed of the 60 performance-monitoring events available in the architectural Performance Monitoring Counters (PMCs). 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 as measured by the performance counters. When a new, unseen program is fed to the model, it first samples the performance counters to make an initial prediction on 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 kept. Because this technique by Cavazos *et al.*relies on iterative compilation, it is not directly applicable to JiT environments, due to the increased compilation cost.

In the work by Dubach *et al.* [12], a machine-learned model based on Artificial Neural-Networks (ANNs) is trained to speed up the process of searching for better optimization transformations for a given program. The model is exposed to different combinations of code transformations and their associated speedups. This way, the iterative process is accelerated by skipping the execution of several versions of the program. Instead, the learned model predicts the speedup of applying a set of code transformations. The model is also capable of predicting the speedup of unseen sets of code transformations. The authors report success in accelerating the iterative compilation process. While the proposed methodology is able to skip most of the iterative compilation process, it requires multiple evaluations using the learned model until the compilation plan is predicted. The cost of these multiple evaluations (at least 16) can outweigh the benefits of the compiled code, when considering a JiT compilation scenario.

Another approach is the automatic tuning of compiler heuristics, discussed in Section 9.2.2.

9.2.2 Automatic Tuning of Compiler Heuristics

Eeckout *et al.* [21] propose an automated compilation-plan tuning approach 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. This work builds on top of previous work by the same authors, in [20], but using GCC as the underlying compiler.

The key difference between the work of Eeckout *et al.* and the approach described in this thesis is that their goal is to adjust the compilation plans in the compiler, which is a coarser approach when compared to method-specific compilation (discussed next in Section 9.2.3). Their methodology has the advantage of not incurring the overhead of a machine-learned model whenever a compilation is carried out. However, once the compilation plans are tuned and deployed, they are bound to encounter the same issues in the beginning of the process. For example, if the compilation plans are tuned considering a set of similar applications, executing applications that are significantly different from the training set can lead to degraded execution performance.

In [39], Vaswani *et al.* propose the use of machine-learned models to predict the execution time of a program given a set of code transformations. The models are built using experimental data and characteristics of the platform (cache sizes and memory latency, for example). Their methodology takes into consideration multi-valued heuristics, such as loop-unrolling and procedure-inlining factors. By including the characteristics of the platform in the learning process, the resulting models are able to perform predictions on a (not significantly) different platform. The training data is collected using a sampled version of a cycle-accurate processor simulator, reducing the generation of data to a few hours with negligible error due to the sampling. The learned model customizes the compiler heuristics for a program or set of programs. GCC was used as the underlying compiler in their study. The authors claim that this methodology significantly simplifies the search and tuning of compilation heuristics, outperforming the default heuristics. Similarly to the approach described in this thesis, the methodology of Vaswani *et al.*focuses on subset of code transformations implemented in GCC. The authors report that the ideal compilation plan varies from program to program, which is consistent with the findings presented in this thesis.

Section 9.2.3 discusses finer-grained approaches for guiding the compilation.

9.2.3 Fine-grained Compilation

Fine-grained compilation approaches adjust the compilation heuristics to smaller portions of the code. How large such portions of code are, depends on the specific approach. Some approaches operate on segments of code, while others consider portions of code at the procedure level (method-specific). Finer-grained approaches consider even smaller portions of code, for example, single loops. The approach described in this thesis is *method-specific compilation*, where a compilation plan is tailored to the method being compiled. The characteristics of the method are summarized by a feature vector, which is then used by the trained machine-learned model to predict the compilation-plan modifier that should be used in the compilation.

Cavazos and O'Boyle [9] trained machine-learned 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. The models indicate the code transformations that should be applied during the compilation of a method, on a per-method basis. 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. The authors 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.

Compared to the approach described in this thesis, the work by Cavazos and O'Boyle is the closest-related work in the literature. However, their work differs from this thesis in several aspects. First, the number of code transformations available in Jikes is significantly smaller than the number of code transformations in Testarossa (in the approach described in this thesis, a subset of 58 code transformations from the available code transformations in Testarossa are controlled).[2] This makes the search for better compilation plans in this study several orders of magnitude larger ($\log(2^{58}/2^{20}) \approx 11.4$), even under the conservative assumption that no more than 50% of the code transformations in a compilation plan will be disabled. Second, Testarossa has an additional optimization level.[3] Third, the dimensionality of the feature vector in this study is considerably larger. On the other hand, a logistic-regression model is capable of outputting compilation plans not seen during the training (since it is essentially a regression technique), as opposed to a multi-class SVM.

Pan *et al.* propose a methodology to divide an input program into segments (tuning sections) automatically [33]. The infrastructure instruments these segments and explores different compilation settings. The compilation settings are rated so that the best settings are retained for each tuning section. The tuning sections are isolated into distinct compilation units that are compiled individually, using source-to-source translation. During the course of the exploration, each code segment uses a customized set of compilation settings, and the final binary is generated using the best set of code transformations for each section, with the instrumentation code removed. This work is a refinement of previous work by the

[2]The exact number of code transformations implemented in Testarossa is confidential.

[3]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 (AoT) compilation), not used in this thesis.

same authors [32]. The scope of that earlier work is, however, enlarged to the complete program being compiled. Because the technique proposed by Pan *et al.* relies on source-to-source translation in order to explore different compilation plans on tuning sections, the technique is not suitable for JiT compilers.

The works by Wu *et al.* [41, 40], having GCC as the underlying compiler, tackle the fact that the highest optimization level frequently leads to suboptimal performance. The methodology builds a knowledge base from training programs using clustering techniques, organizing them with respect to code segments. The methodology described is applied only to loops found in the intermediate representation of the program, even though there is no limitation to consider other segments of the code. The model is trained considering optimization settings for similar code segments. The similarity is given by their feature vectors. When the model is learned, new program inputs are segmented in order to search for similar segments in the knowledge base, and the associated compilation settings are used. The authors report consistent speedups over the default heuristics in GCC, including the highest optimization level.

The key difference to the approach described in this thesis is that the methodology proposed by Wu *et al.* is able to determine when the code segment being compiled is significantly different from those known to the learned model, because it uses a machine-learning algorithm based on clustering. Therefore, their methodology can quantify the distinctiveness of the code portion by computing the distance of the respective feature vector to the nearest cluster in the feature space. In this case, the conservative approach of relying on the default heuristics is taken. In the approach described in this thesis, the default heuristics are chosen only if the learned SVM predicts the class for the *null compilation-plan modifier*.

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 what compiler designers consider to be descriptive features might not be ideal from the machine-learning algorithm perspective. Leather *et al.* [27] proposed a methodology to automatically create features based on grammar evolution. Simple grammar constructs are made available to the evolutionary algorithm, which creates rules that can be evaluated over the intermediate representation of a program being compiled. The features are evolved over several generations, and the most descriptive features are kept to train a machine-learned model. The underlying compiler used is GCC, and the authors report significant improvements over the default heuristics in GCC, including the maximum optimization level.

While the automatic methodology to generate program features is effective, the cost is significant. First, the methodology proposed by Leather *et al.* requires an additional data-collection process and experimental evaluation in order to automatically derive a set of features, until the evolutionary algorithms converge to a solution. The methodology evaluates whether a possible set of features improves the accuracy of the model, thus requiring the training and evaluation of several models. Second, considering that such evolutionary process is likely to be performed using smaller-scale data collections, an additional, large-scale data-collection process is still necessary to generate training data for the machine-learned model that will be deployed with the compiler. Finally, the evolutionary approach can generate features that are computationally intensive, up to 2 s of CPU time (if a feature takes more than 2 s of CPU time to evaluate, it is not included in the next generation). In the platform used in this thesis, Testarossa rarely takes more than 250 ms to compile a method at the highest optimization level. Therefore, the automated approach to generate features proposed by Leather *et al.*, in its current form, is not viable for JiT environments.

Chapter 10

Conclusion

This thesis describes the design and implementation of a complete and customized framework that enables the use of machine-learned models for method-specific compilation in Testarossa, a commercial JiT compiler from IBM. The thesis motivates the need for such a customized infrastructure, in order to overcome challenges from production environments, such as large-scale data collection.

The framework is composed of (*i*) a data collection infrastructure that implements a lightweight profiling mechanism; (*ii*) a binary archive format for handling large-scale data collection; (*iii*) tools for processing the data collected in order to train machine-learned models; and (*iv*) a lightweight communication protocol for integrating the compiler and the machine-learned models.

The data-collection process described in this thesis can generate a significant amount of training data in a few hours, and can be performed in parallel on a computer cluster.[1] The multiple approaches to generate compilation-plan modifiers are complementary, and the resulting data can be easily merged in multiple ways. For instance, the leave-one-out cross-validation process benefits from the ability of merging multiple distinct data sets.

The customized binary archive format enables large-scale data-collection experiments to be conducted. The archives store approximately $7,000$ compilation records per megabyte, without compression.[2] A reader application was implemented, allowing the data collected to be easily interfaced with other tools, especially those shipped with the implementation of machine-learning algorithms, such as LIBLINEAR.

The tools implemented for the ranking process can handle large amounts of data and generate reasonably-sized training data sets. Currently, the size of the training data set is proportional to the number of unique methods present in the data processed during the ranking process. The size of the data sets can be further controlled if necessary, by either

[1]Data collection for the 5 SPECjvm98 benchmarks, running on a single node of the cluster used in this study, completes in approximately 20 hours.

[2]The archives can be compressed, further increasing the storage savings (the compression ratio ranges from 40% to 80%).

using stricter parameters in the ranking process, or using additional trimming tools that were also implemented. The ranking process was improved during many iterations, and training times are down from several days to a few minutes, requiring significantly less resources.[3] The ranking process can also benefit from multi-processor and/or clustered environments, if resources are available.

This thesis also describes a simple and low-overhead integration mechanism to interface with different implementations of machine-learned models. This mechanism enables the use of different machine-learned models without changes to the compiler. This mechanism also allows for easy evaluation of different implementations of machine-learned models to find one that is appropriate for production environments.

The approach proposed in this thesis does not outperform the performance throughput of Testarossa when operating in its default mode (adaptive, multi-level optimization). However, the performance results provide evidence that, for some types of benchmarks, method-specific compilation plans produce code comparable to that generated by the plans currently used in Testarossa. These plans were developed over many years by expert developers.

On the other hand, when using the machine-learned models based on data from the randomized search approach (Section 5.3 on page 39), including the hybrid models, the compilation times are significantly smaller. In particular, models created using data collected solely from the randomized search approach produce compilation times significantly smaller than Testarossa operating in adaptive mode, and much smaller than Testarossa using only the *hot* optimization level. The models using progressive randomized search (Section 5.4 on page 40) are not able to significantly reduce compilation time because they focus their search on plans similar to the original ones included in Testarossa.

Chapter 11 discusses areas for future development of the infrastructure presented in this thesis.

[3]The training times of each of the models discussed in this thesis range from 2 to 40 minutes on a 1 GHz laptop, depending on the value of the misclassification cost parameter C for SVMs.

Chapter 11

Future Work

JiT compilers offer a challenging compilation environment. They also offer numerous possibilities for research in order to convert a compilation budget into improved execution time for an application.

This chapter discusses future work in the context of this thesis, in a logical sequence. The first of these possible directions is feedback during the data-collection process, discussed in Section 11.1.

11.1 Active Feedback Exploration

The approaches implemented for generation of compilation-plan modifiers are, by design, passive in the sense that once they are generated, they do not change in any way. The main advantage is the reduced complexity because only a few variables unrelated to the application being executed are taken into consideration. However, there is no guarantee that the exploration is being focused in the most relevant areas of the feature space for an arbitrary method.

To improve the search process, modifiers can be generated on-demand after receiving feedback from the instrumented application. This can be accomplished by computing the ranking value of a modifier for a method after it executes for a certain number of invocations, and then deciding what should be the next compilation-plan modifier. The ranked modifier serves as a hint for generating modifiers, which can focus the future modifiers into specific areas of the modifier space.

Using this feedback-based approach, modifiers are generated on a per-method basis. Therefore, the exploration of the optimization space can be focused on areas of the optimization space that are relevant for the method. For example, modifiers for methods containing loops can focus on areas of the optimization space concerning loop transformations, provided that those transformations are beneficial to the method (*i.e.*, an improvement in the execution time can be measured).

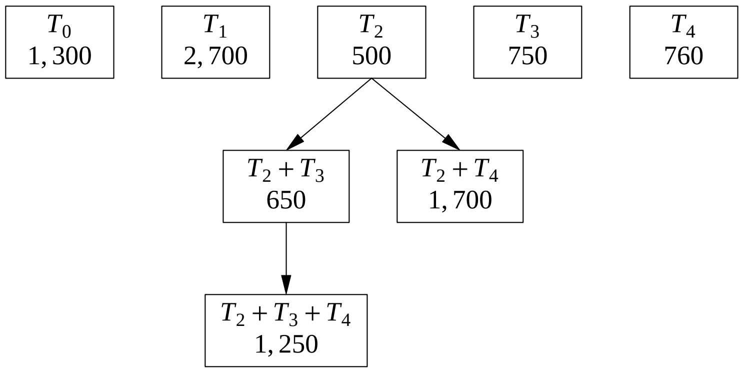

Figure 11.1: Heuristic search of compilation plans. The boxes contain a set of code transformations and their respective ranking value, for an arbitrary feature vector. The search focuses on regions with lower cost (as given by the ranking value), including code transformations based on their ranking value from the initial state (the uppermost level).

The communication can reuse the socket-based infrastructure in place, which is used to communicate with the machine-learned model. As far as Testarossa is concerned, there is no difference between a compilation-plan modifier generated based on runtime feedback and a modifier predicted by a learned model.

This feedback-based approach can also enable the use of different machine-learning algorithms. In particular, algorithms that incrementally adapt to changes in the learning scenario, such as Reinforcement Learning (RL) or *incremental* SVMs [14] might be considered.

Section 11.2 discusses a pruning approach to be used in conjunction with active feedback exploration.

11.2 Heuristic-based Search

Due to the size of the optimization space, searching for different compilation plans can be very time-consuming. Moreover, because of the method-specific approach, the size of the optimization space to search is enlarged across each distinct feature vector. In order to focus the search on the portions of the optimization space that are relevant to the feature vector, a heuristic-based search can be used.

For the heuristic-search algorithm, the state of the search is represented using the ranking value for a given feature vector and the respective compilation-plan modifier. The initial step in the search is, therefore, to enable each individual code transformation in order to assess the sensitivity of the feature vector regarding each of the code transformations.

Figure 11.1 illustrates the state-transition during the search. The transition is based on the ranking values obtained in the initial state. The heuristic-search algorithm focuses on the regions of the optimization space with smaller costs, given by the ranking function described in this thesis. As the search progresses, the algorithm generates compilation-plan modifiers by combining an additional code transformation in a previous state of the search (*i.e.*, by including a code transformation in a previous compilation-plan modifier). In the second level of the figure, the algorithm evaluates the inclusion of two code transformations, generating the distinct sets $\{T_2 + T_3\}$ and $\{T_2 + T_4\}$. The transformations T_3 and T_4 were chosen because of their low costs as discovered in the initial step, in the uppermost level of the figure. For the third level, the algorithm evaluates a set of code transformations combining $\{T_2 + T_3 + T_4\}$.

The heuristic-based approach dynamically explores the most yielding regions of the optimization space. This process must be performed for every unique feature vector. On the other hand, the approaches implemented in this thesis to generate compilation-plan modifiers do not take any additional variables, thus they randomly sample the optimization space. Depending on the characteristics of the method, such randomized approaches can evaluate numerous modifiers that are not relevant for a given method.

Section 11.3 discusses a method for accelerating the search (regardless of the approach used to generate compilation-plan modifiers), taking the similarity of feature vectors into consideration and the previously modifiers used.

11.3 Accelerated Search based on Similarity

The search for compilation-plan modifiers requires the exploration of a large space of possible modifiers (2^{58}) for each unique feature vector. If feature vectors can be identified during the data-collection process, and if they are similar enough (*i.e.*, sufficiently close in the feature space), these similar feature vectors can benefit from the search efforts invested previously.

However, because of the dimensionality of the feature vectors currently in use (71-dimensional), small differences still result in unique feature vectors. An unsupervised learning algorithm based on *clustering* (*e.g.*: k-Nearest Neighbors) can be used to identify sufficiently similar feature vectors. In this case, distinct methods with highly similar feature vectors are considered as equal for exploration of the optimization space.

The similarity function used in the clustering algorithm, which measures the difference between two feature vectors, must take into consideration the contribution of each component of the feature vector. For example, two methods that differ only by an attribute such as *has loops* cannot be merged together. This is because the performance of a method containing loops is greatly reduced if no code transformations for loops are used during the compilation. On the other hand, feature vectors differing only an arbitrary amount in the number of locals (*i.e.*, variables used in the body of the method) can be safely merged.

This similarity-based approach for accelerated search can enable the effective exploration of ordered compilation plans, which is discussed in Section 11.4.

11.4 Ordered Compilation Plans

The natural step in method-specific compilation is the generation of ordered compilation-plans. An ordered compilation-plan specifies precisely *what* code transformations to apply during the compilation, and in *which order*. The order in which code transformations are applied can either create or hinder optimization opportunities during the compilation.

Currently, ordered compilation-plans are manually crafted by expert compiler developers, and tuned over the course of many years. Maintaining such compilation plans is a daunting task, especially when considering the diversity of code that is compiled and the fast-paced changes in supported platforms.

The compilation-plan modifiers described in this thesis only specify what code transformations to apply, without changing the order in which they are applied by Testarossa. The ability to create ordered compilation-plans can further exploit the benefits of method-specific compilation. Moreover, distinct methods benefit differently from compilation plans. For example, for a method that benefits from a compilation plan where many code transformations are disabled, it is possible that such a compilation plan can be streamlined further. In practice, it is common for code transformations to be applied more than once during the compilation, especially clean-up transformations. In such case, the number of clean-up passes can be reduced, or even removed altogether if the method is not sensitive to them (*e.g.*: the method does not have unreachable portions of code, so code transformations for dead-code elimination are not as effective). On the other hand, computationally-intensive methods can benefit from longer compilation plans, which are not appropriate for a number of methods that are not as computationally intensive.

Traditional machine-learning algorithms can be used with ordered compilation-plans by reusing the mapping approach used to circumvent the class-space limitation in LIBLINEAR. In this case, each distinctly ordered compilation-plan is assigned an unique identifier, which is used as a class for the training data set. When used by Testarossa, a predicted class is mapped back to the original ordered compilation-plan.

Bibliography

[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] Advanced Micro Devices. *AMD64 Architecture: Programmer's Manual Volume 2: System Programming*, November 2009.

[3] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Code Generation and Optimization (CGO)*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[5] 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.

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

[7] 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.

[8] John Cavazos and J. Eliot B. Moss. Inducing heuristics to decide whether to schedule. In *Programming Language Design and Implementation (PLDI)*, pages 183–194, Washington, DC, June 2004.

[9] 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.

[10] Michał Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java^{TM} Under Dynamic Optimizations. *Programming Language Design and Implementation (PLDI)*, 35(5):13–26, 2000.

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

[12] 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.

[13] 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.

[14] Glenn Fung and Olvi L. Mangasarian. Incremental Support Vector Machine Classification. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, pages 247–260, 2002.

[15] 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.

[16] 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.

[17] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley Professional, 2005.

[18] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. Java$^{\mathrm{TM}}$ 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.

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

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

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

[22] IBM Corporation. IBM J9$^{\mathrm{TM}}$ Java$^{\mathrm{TM}}$ Virtual Machine. `http://www.ibm.com/developerworks/java/jdk/`.

[23] IBM Corporation. Performance Inspector$^{\mathrm{TM}}$. `http://perfinsp.sourceforge.net/`, July 2010.

[24] 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.

[25] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(2):165–198, 2005.

[26] 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.

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

[28] Robert Love. *Linux Kernel Development*. Novell Press, 2005.

[29] Jean-loup Gailly Mark Adler. zlib. `http://www.zlib.net/`, July 2010.

[30] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine Learning Approach to Automatic Production of Compiler Heuristics. *Artificial Intelligence: Methodology, Systems, and Applications*, pages 389–409, 2002.

[31] J. Eliot B. Moss, Paul E. Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla E. Brodley, and David T. Scheeff. Learning to Schedule Straight-Line Code. In *Neural Information Processing Systems (NIPS)*, pages 929–935, Denver, CO, 1997.

[32] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Code Generation and Optimization (CGO)*, pages 319–332, New York, NY, 2006.

[33] Zhelong Pan and Rudolf Eigenmann. Fast, automatic, procedure-level performance tuning. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 173–181, Seattle, Washington, 2006.

[34] Standard Performance Evaluation Corporation (SPEC). SPEC JVM98 benchmarks. `http://www.spec.org/jvm98/`.

[35] Mark Stephenson and Saman Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *Code Generation and Optimization (CGO)*, pages 123–134, San Jose, CA, March 2005.

[36] Mark Stephenson, Saman Amarasinghe, Martin C. Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Programming Language Design and Implementation (PLDI)*, pages 77–90, San Diego, CA, June 2003.

[37] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX® Environment*. Addison-Wesley Professional, 2005.

[38] The Jikes RVM Project. Jikes RVM. `http://jikesrvm.org/`.

[39] Kapil Vaswani, Matthew J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. Microarchitecture Sensitive Empirical Models for Compiler Optimizations. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:131–143, 2007.

[40] Haiping Wu, Eunjung Park, Mihailo Kaplarevic, Yingping Zhang, Murat Bolat, Xiaoming Li, and Guang R. Gao. Automatic Program Segment Similarity Detection in Targeted Program Performance Improvement. *International Parallel and Distributed Processing Symposium*, 0:452, 2007.

[41] Haiping Wu, Eunjung Park, Mihailo Kaplarevic, Yingping Zhang, Xiaoming Li, and Guang R. Gao. Dynamic optimization option search in GCC. In *GCC Developers Summit*, pages 165–174, 2007.

Appendix A

SPECjvm98 Results

The performance results obtained when using the learned models are constrasted with Testarossa operating under adaptive mode, and also using only the *hot* optimization level. The running time of each benchmark is detailed on a cumulative, per-iteration basis. For each benchmark application, a set of three graphs are presented, focusing on each of the three approaches used to generate compilation-plan modifiers.

All running times are reported in seconds. The benchmarks are executed 30 times, and the average time spent on each of the 10 internal benchmark iterations is plotted.

A.1 `compress`

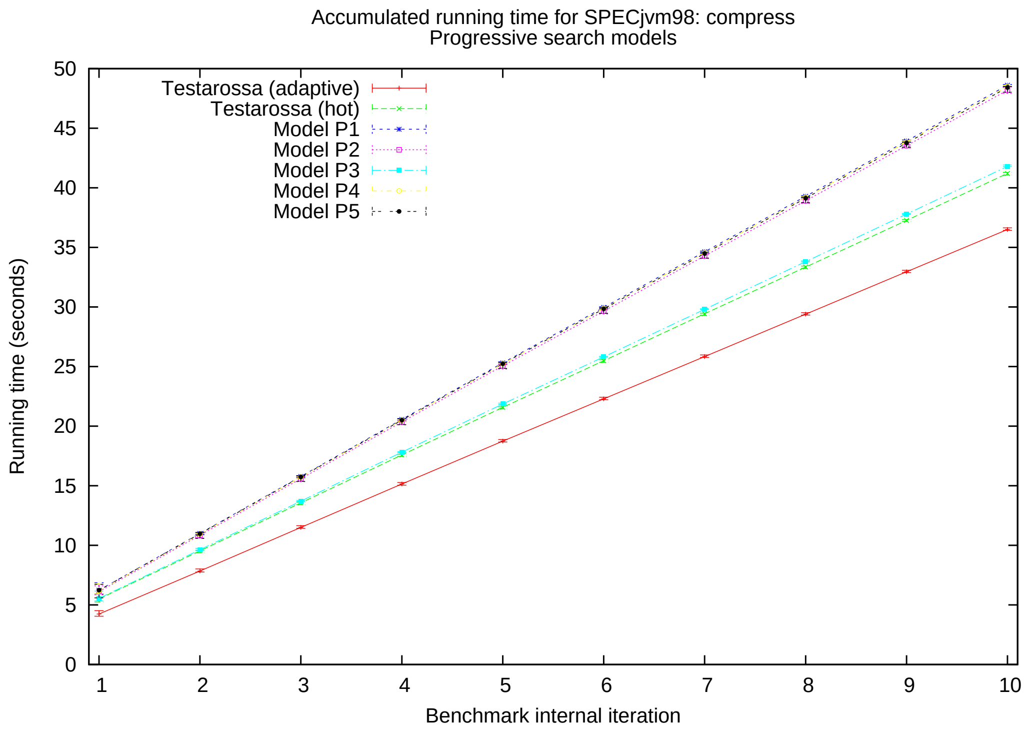

Figure A.1: Progressive randomized search models on `compress`.

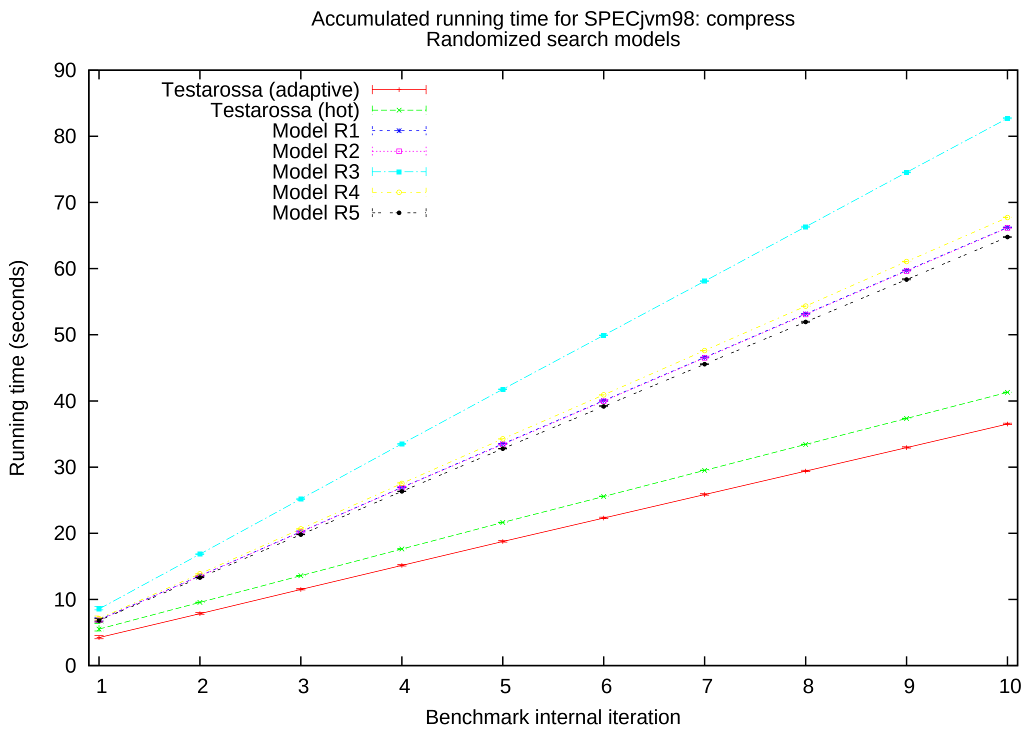

Figure A.2: Randomized search models on `compress`.

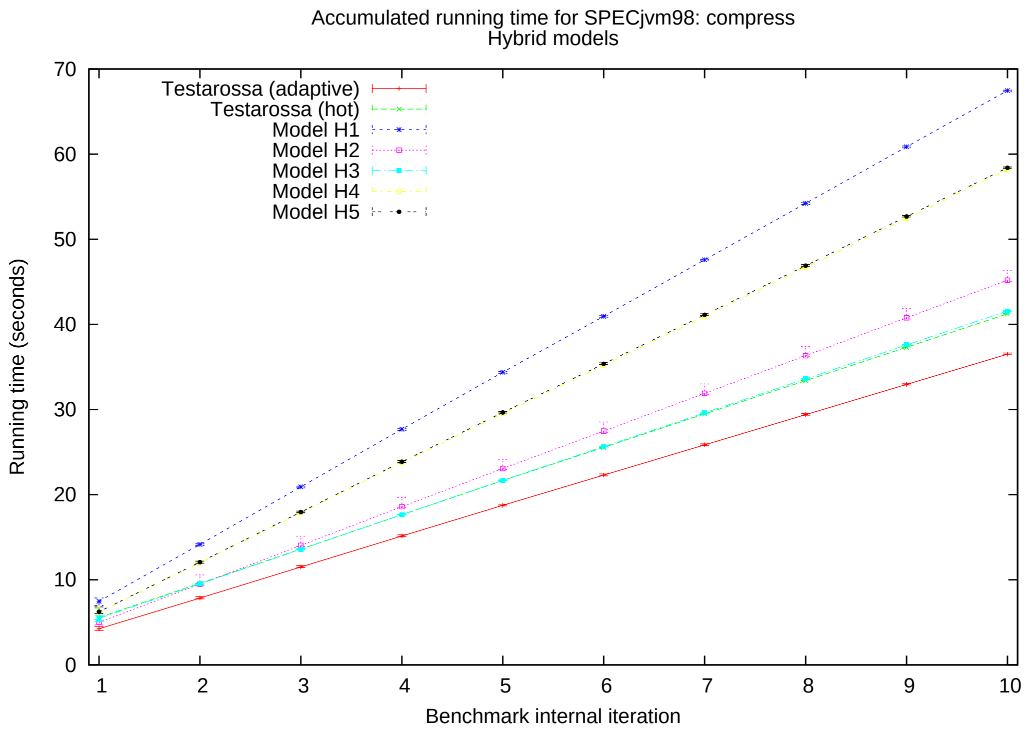

Figure A.3: Hybrid models on `compress`.

A.2 db

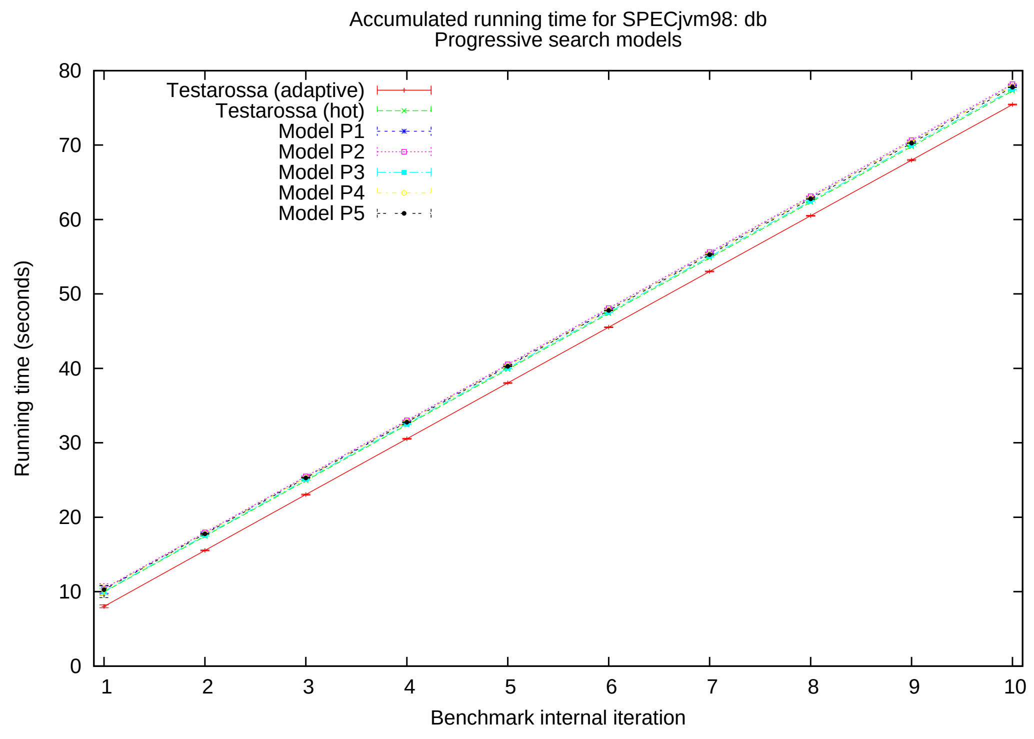

Figure A.4: Progressive randomized search models on db.

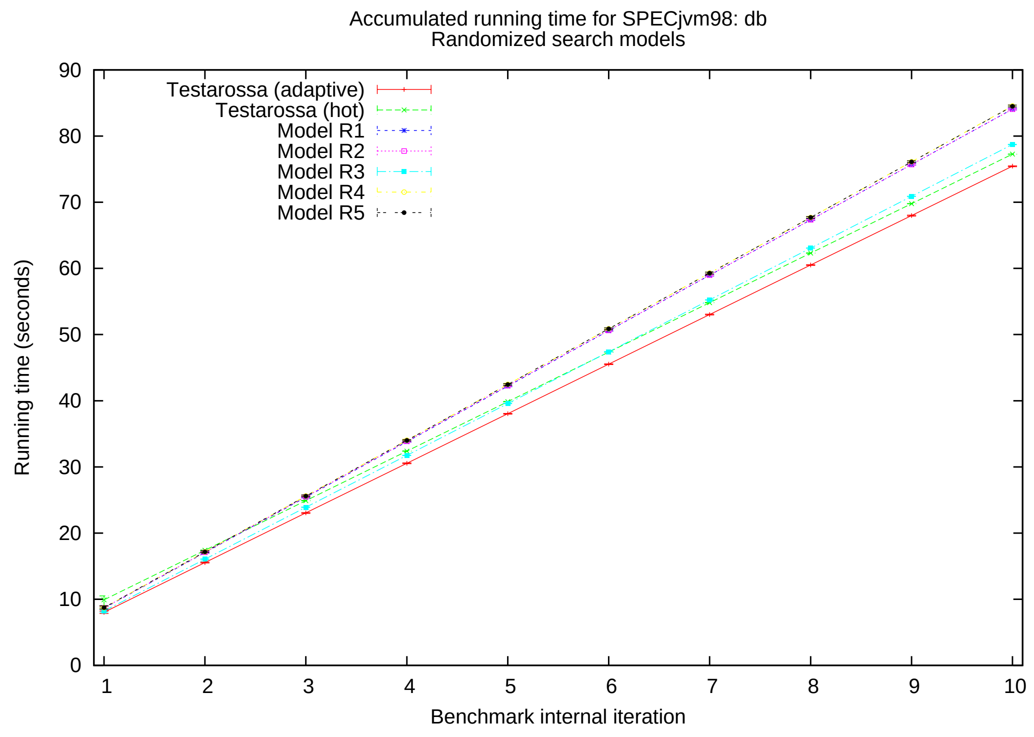

Figure A.5: Randomized search models on db.

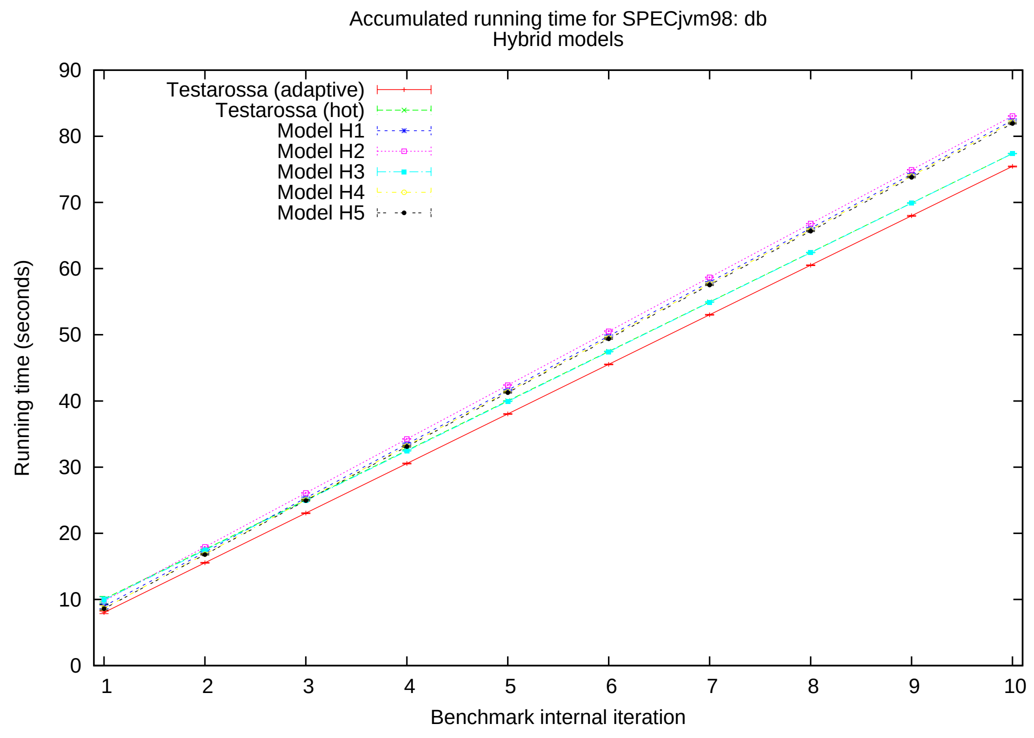

Figure A.6: Hybrid models on `db`.

A.3 `jack`

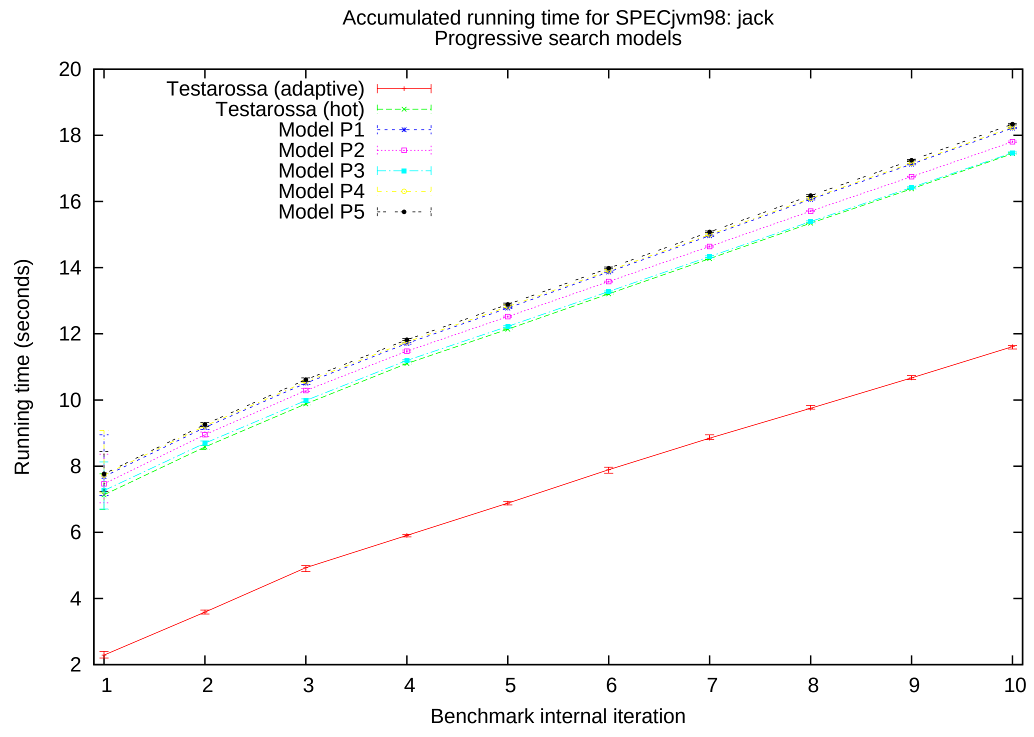

Figure A.7: Progressive randomized search models on `jack`.

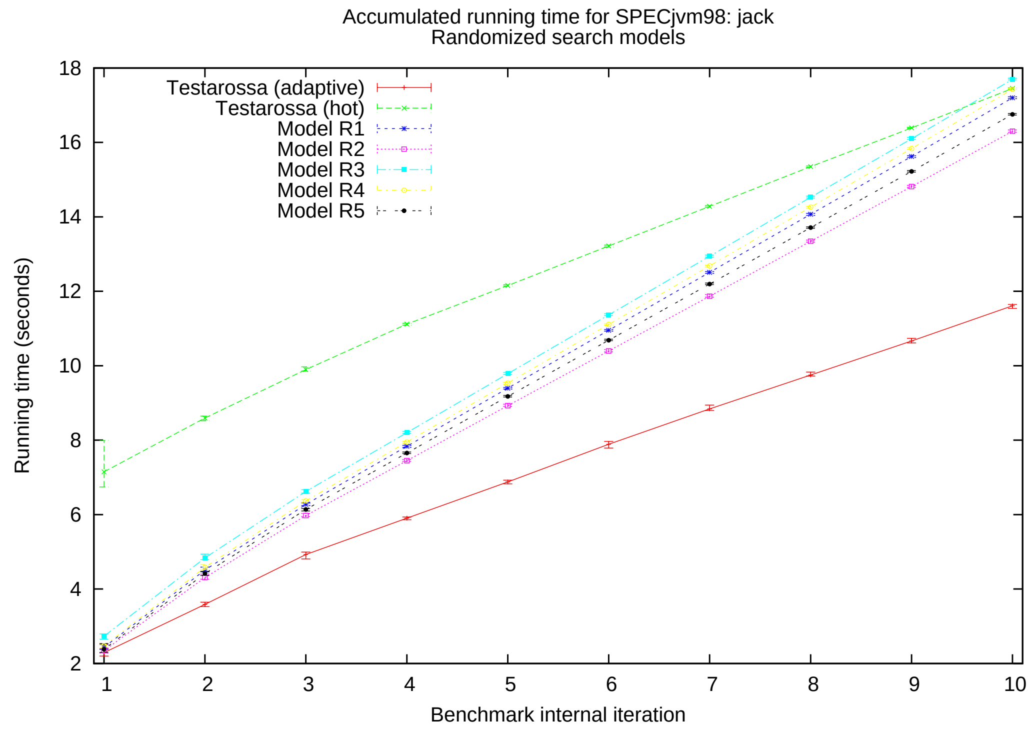

Figure A.8: Randomized search models on `jack`.

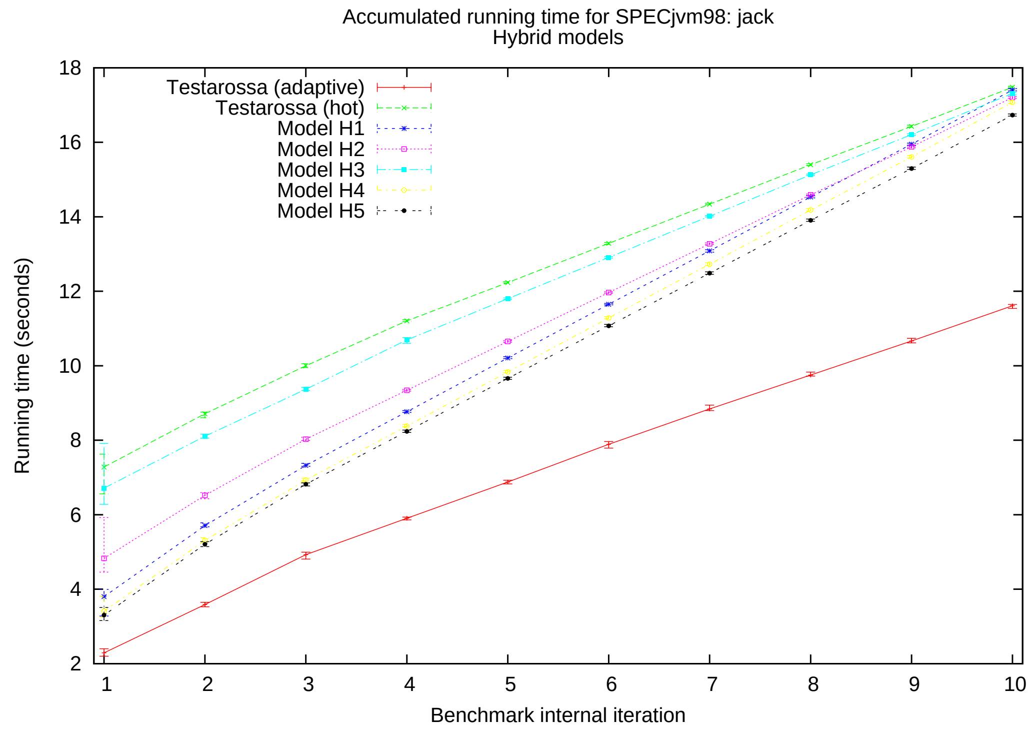

Figure A.9: Hybrid models on `jack`.

A.4 javac

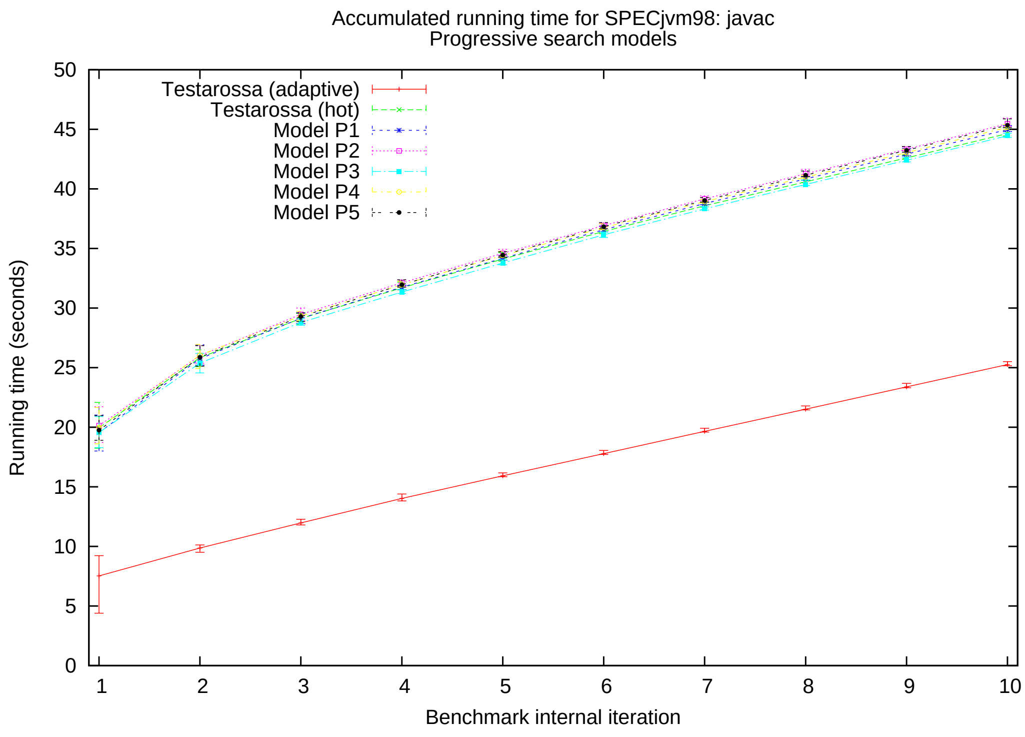

Figure A.10: Progressive randomized search models on `javac`.

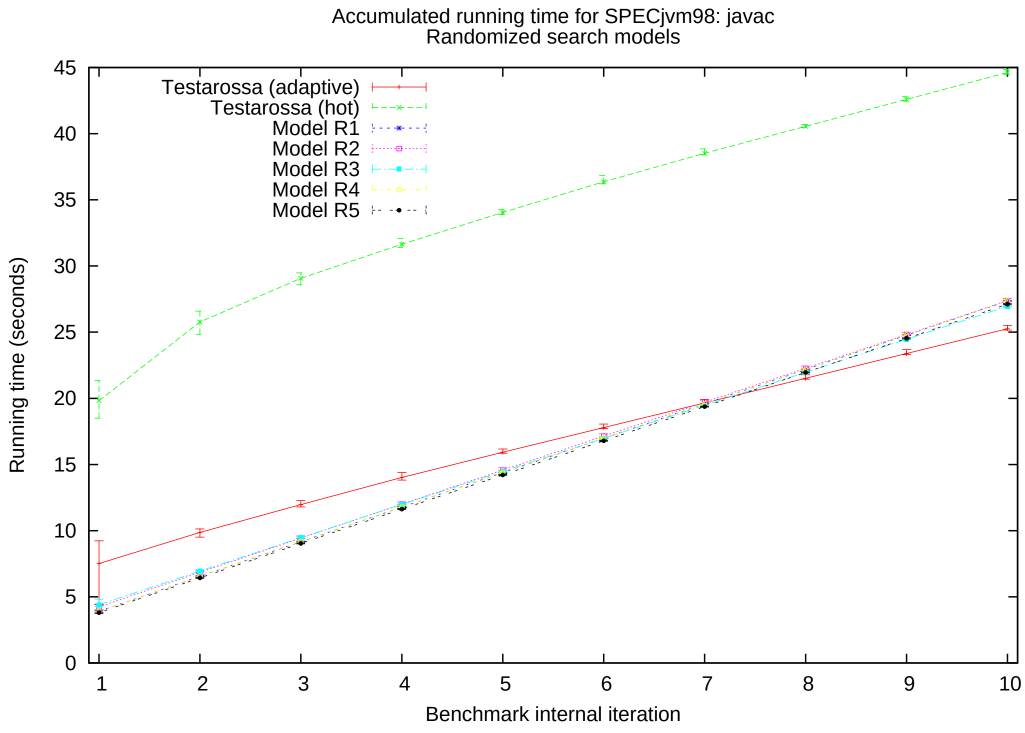

Figure A.11: Randomized search models on `javac`.

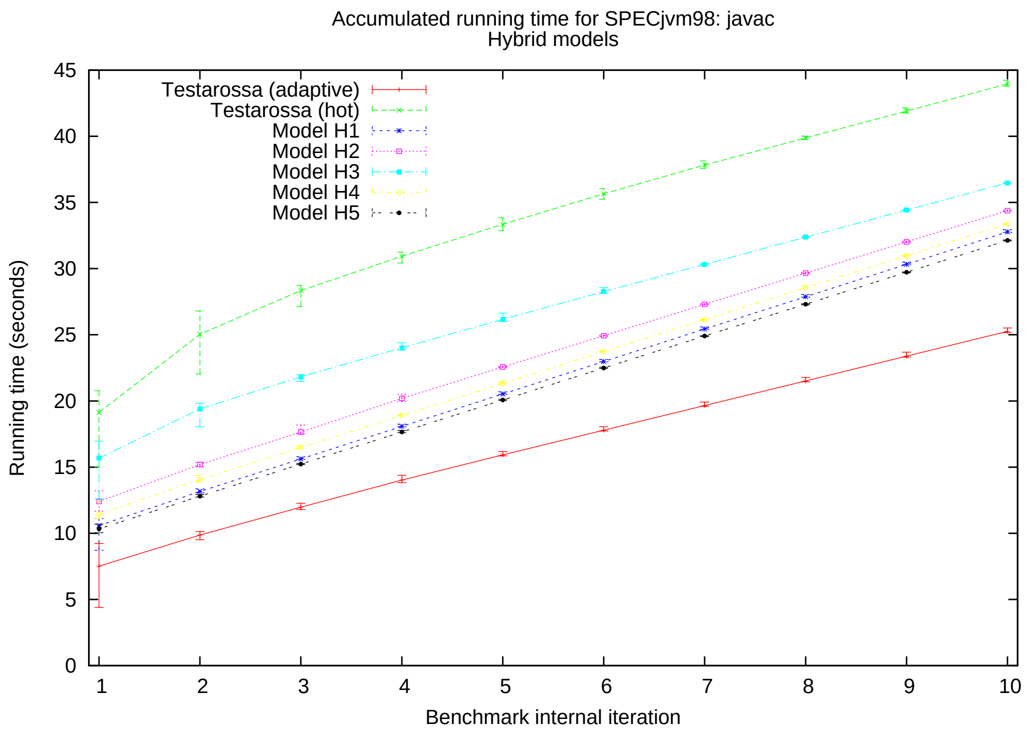

Figure A.12: Hybrid models on `javac`.

A.5 `jess`

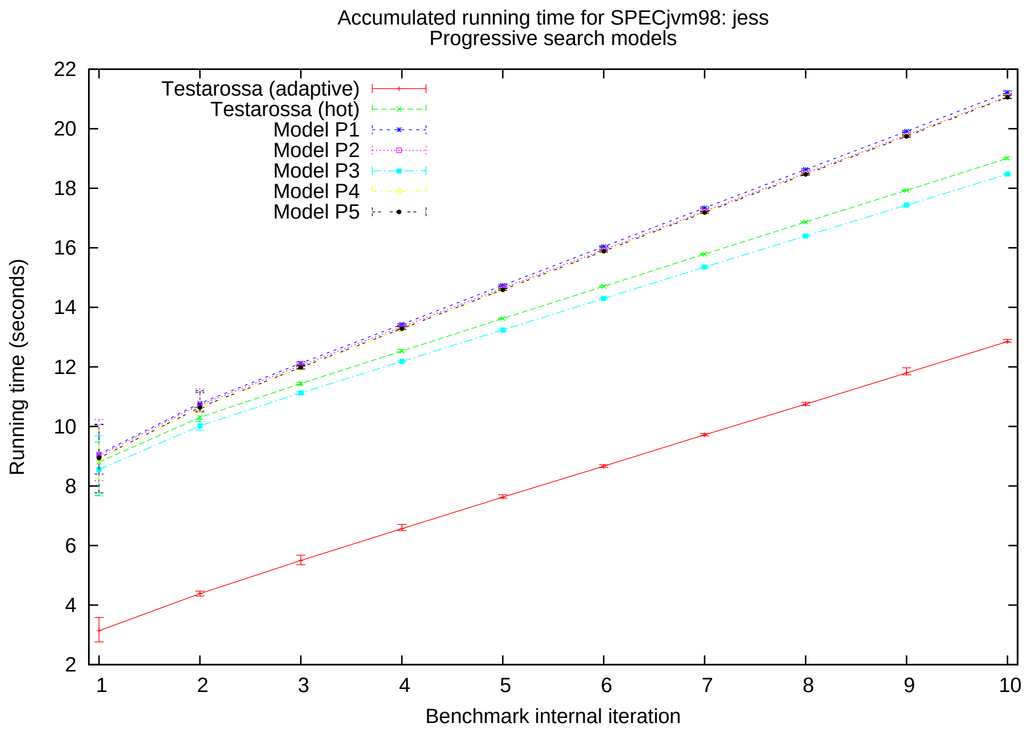

Figure A.13: Progressive randomized search models on `jess`.

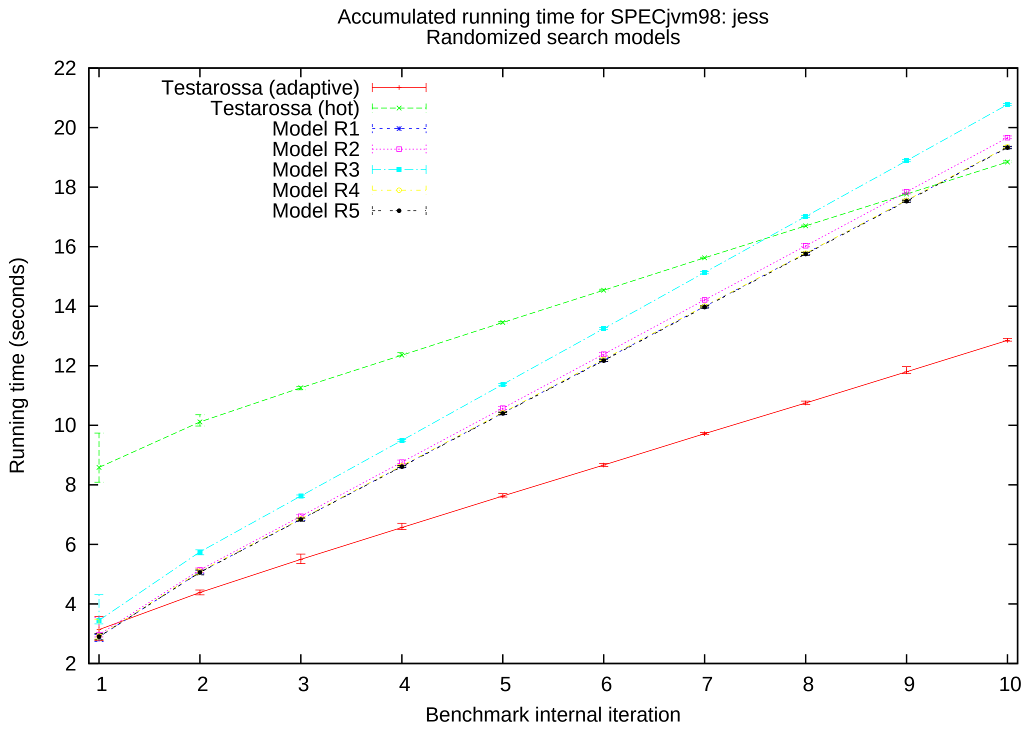

Figure A.14: Randomized search models on `jess`.

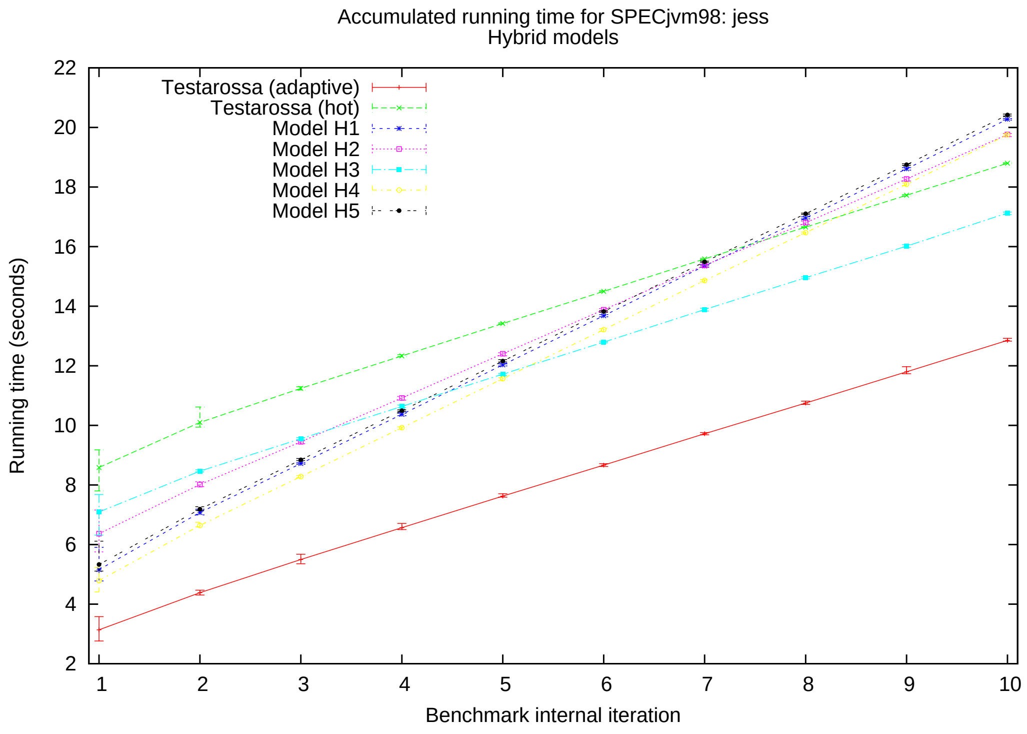

Figure A.15: Hybrid models on `jess`.

A.6 mpegaudio

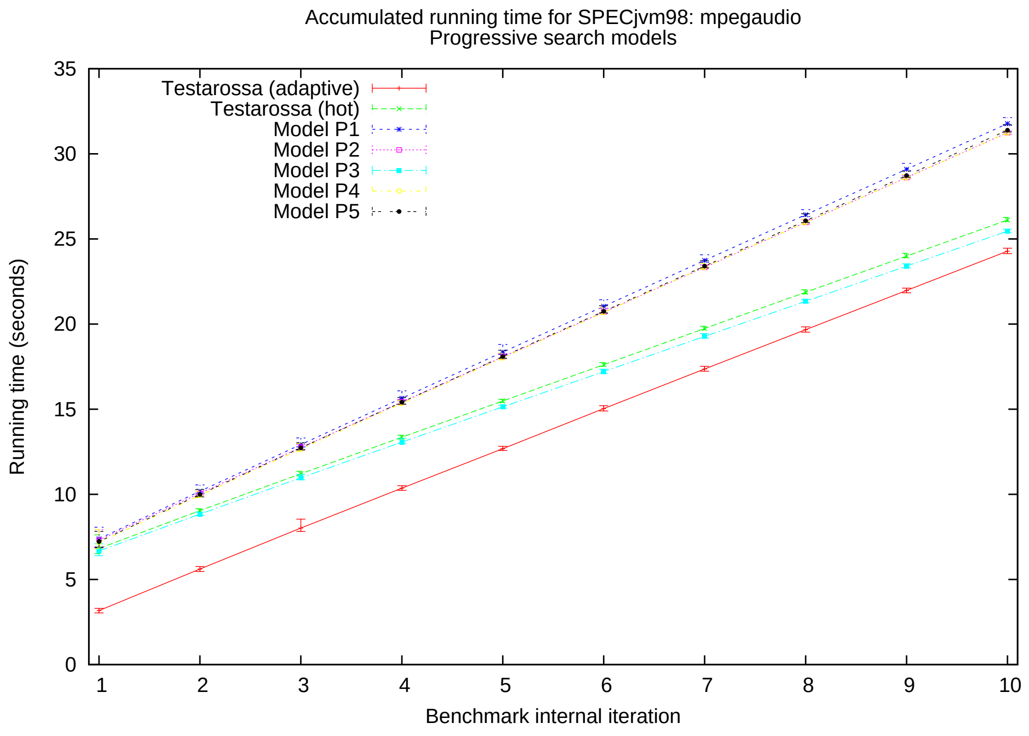

Figure A.16: Progressive randomized search models on `mpegaudio`.

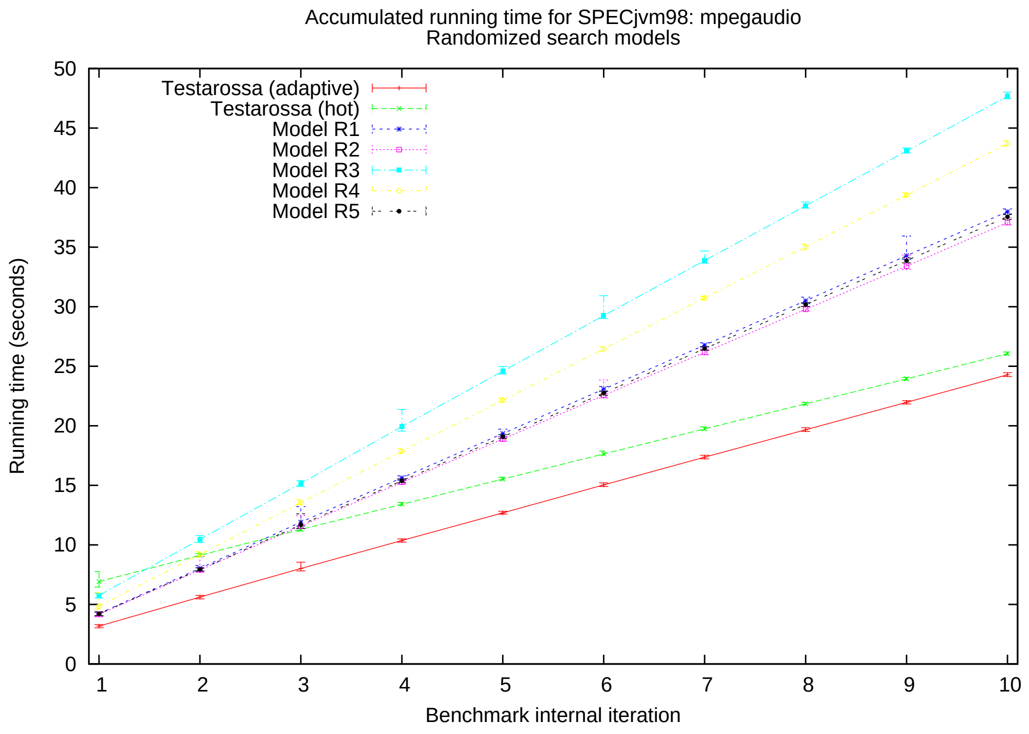

Figure A.17: Randomized search models on `mpegaudio`.

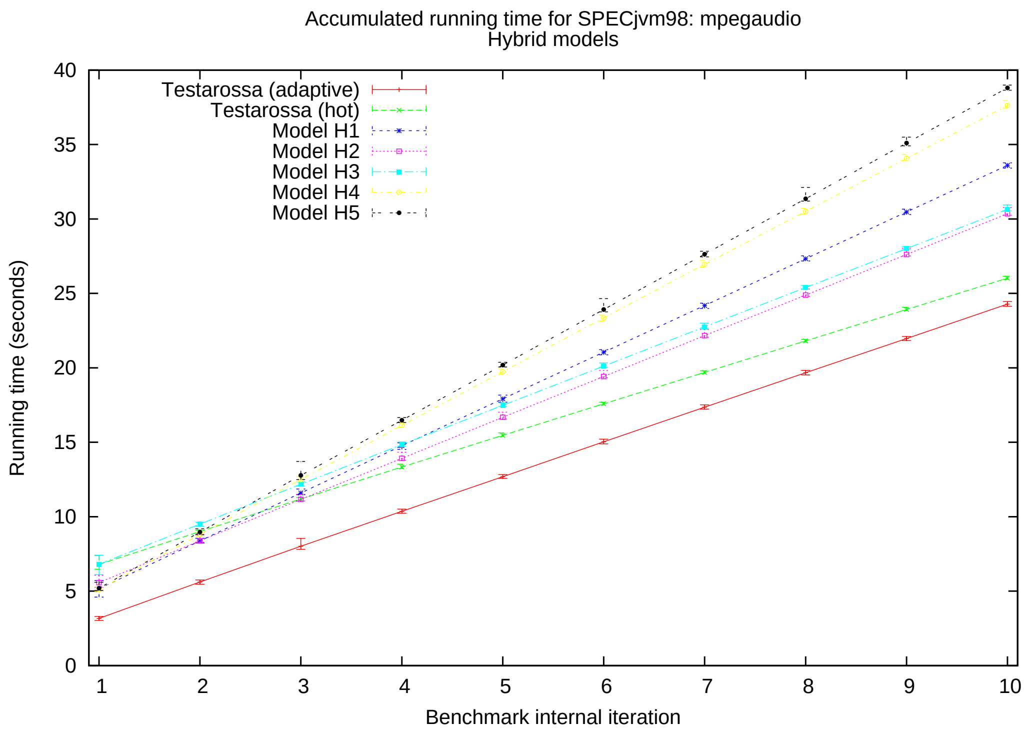

Figure A.18: Hybrid models on `mpegaudio`.

A.7 `mtrt`

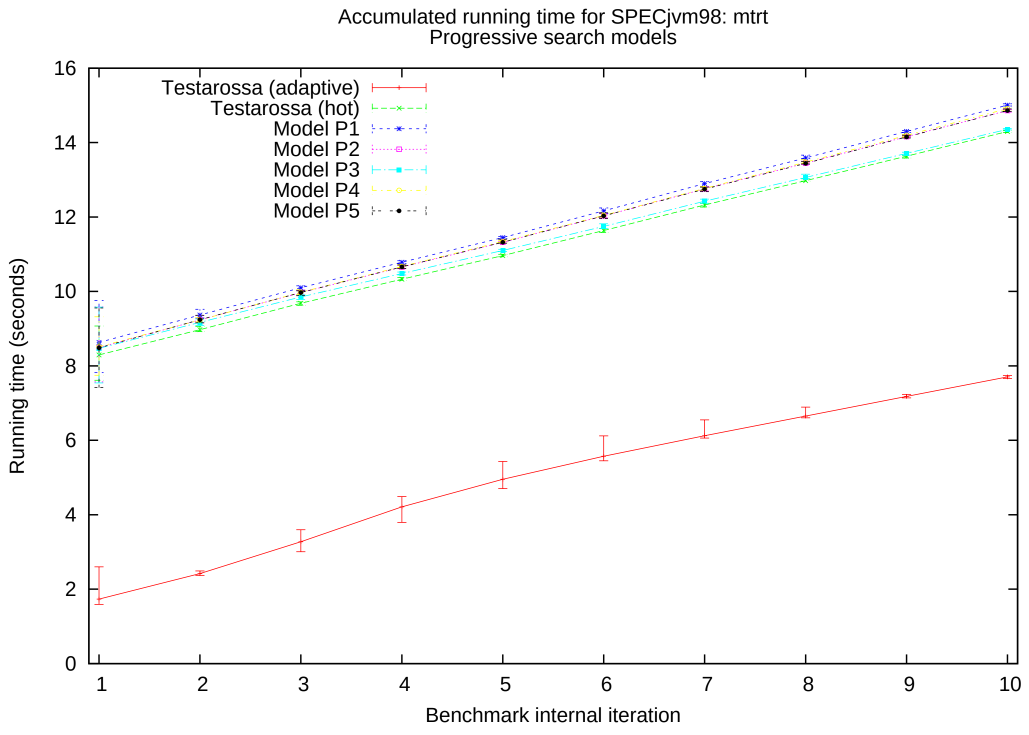

Figure A.19: Progressive randomized search models on `mtrt`.

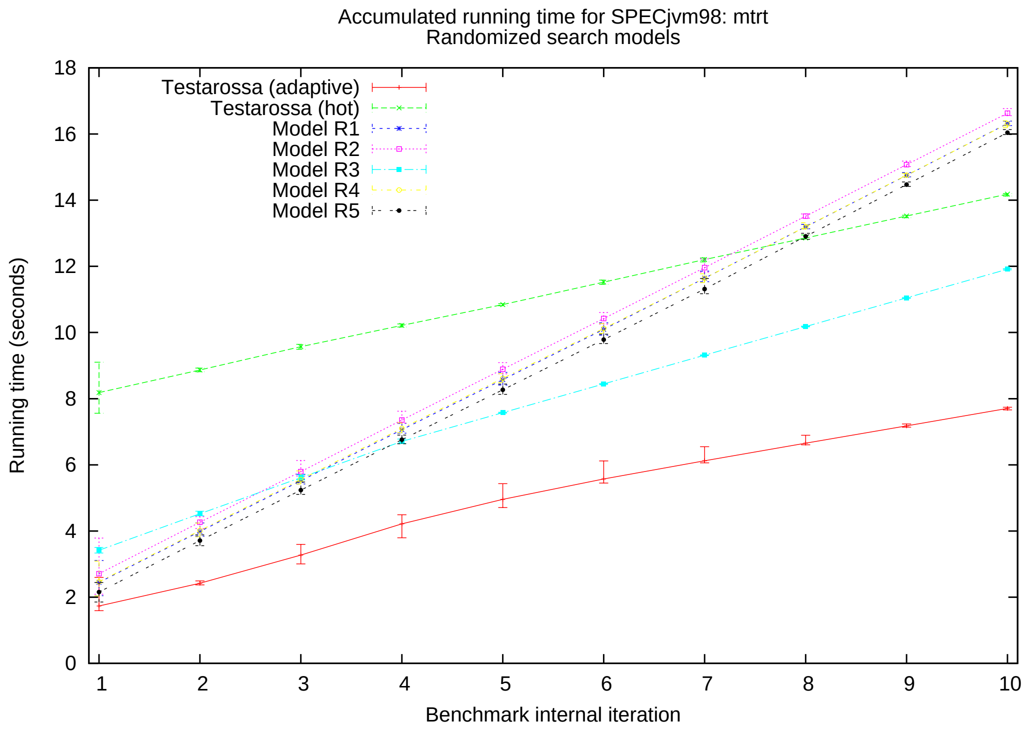

Figure A.20: Randomized search models on `mtrt`.

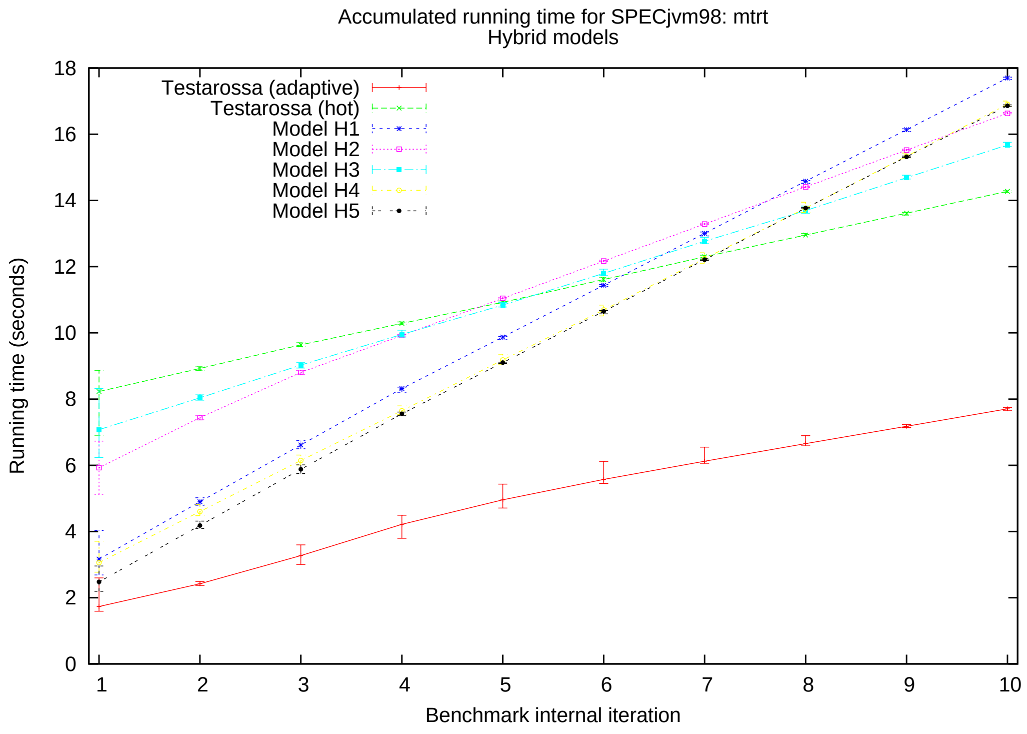

Figure A.21: Hybrid models on `mtrt`.

A.8 `raytrace`

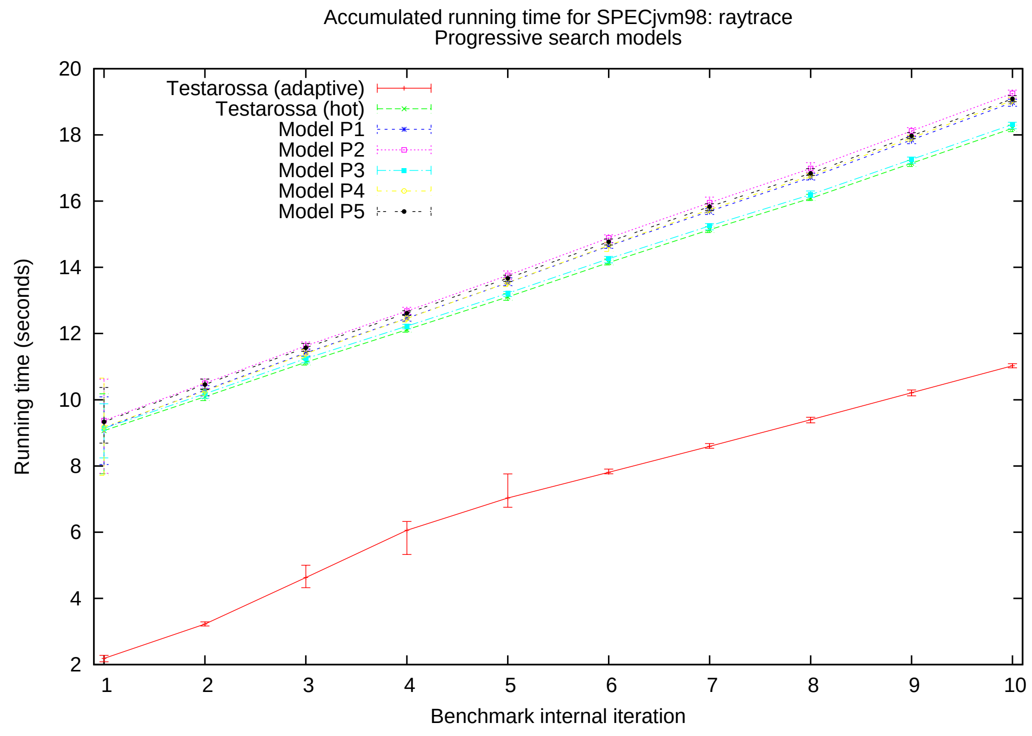

Figure A.22: Progressive randomized search models on `raytrace`.

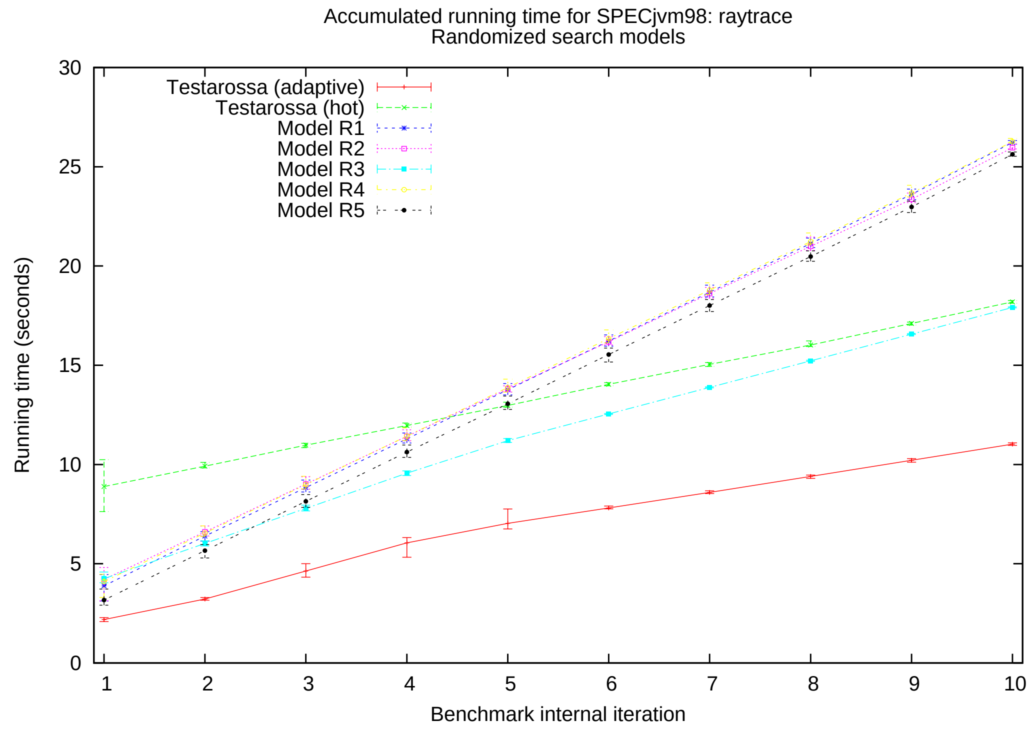

Figure A.23: Randomized search models on `raytrace`.

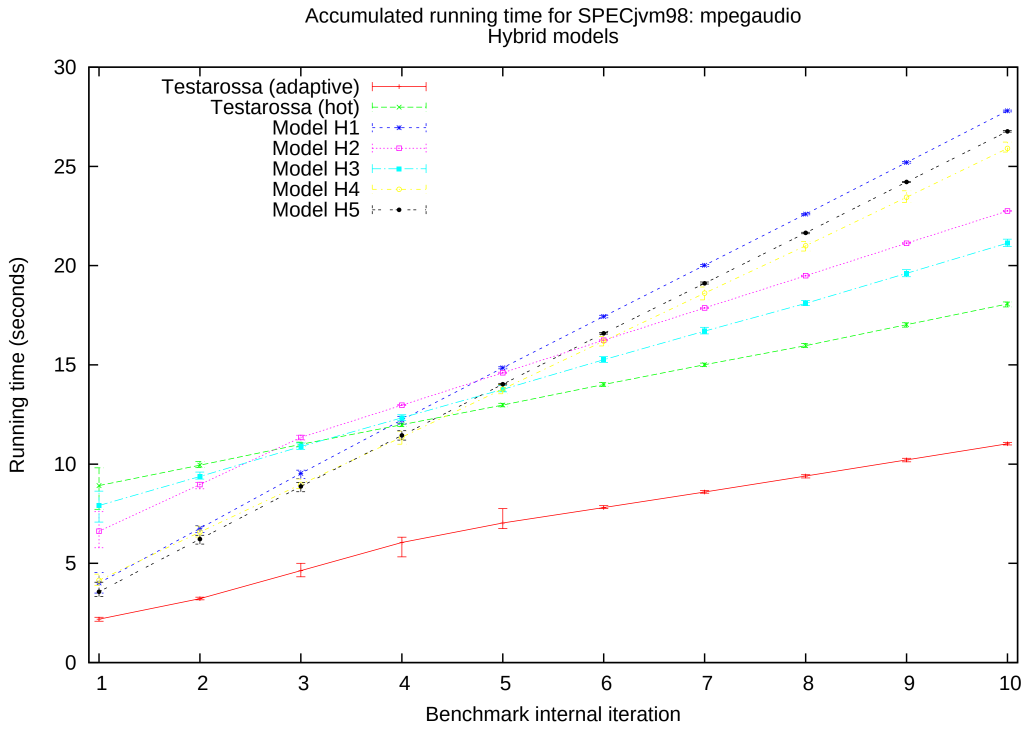

Figure A.24: Hybrid models on `raytrace`.