Collaborative Compilation

by

Benjamin R. Wagner

Submitted to the

Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 2006

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by .... [illegible]
Saman P. Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Collaborative Compilation

by

Benjamin R. Wagner

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Modern optimizing compilers use many heuristic solutions that must be tuned empirically. This tuning is usually done "at the factory" using standard benchmarks. However, applications that are not in the benchmark suite will not achieve the best possible performance, because they are not considered when tuning the compiler. Collaborative compilation alleviates this problem by using local profiling information for at-the-factory style training, allowing users to tune their compilers based on the applications that they use most. It takes advantage of the repeated compilations performed in Java virtual machines to gather performance information from the programs that the user runs. For a single user, this approach may cause undue overhead; for this reason, collaborative compilation allows the sharing of profile information and publishing of the results of tuning. Thus, users see no performance degradation from profiling, only performance improvement due to tuning.

This document describes the challenges of implementing collaborative compilation and the solutions we have developed. We present experiments showing that collaborative compilation can be used to gain performance improvements on several compilation problems. In addition, we relate collaborative compilation to previous research and describe directions for future work.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor

Acknowledgments

I thank the members of the Compilers at MIT (Commit) group, led by Professor Saman Amarasinghe, for their valuable input to this project, especially William Thies and Rodric Rabbah. I also acknowledge the challenging academic environment of MIT and the support of many friends, acquaintances, staff, and faculty during my time at the Institute.

This thesis describes a project that was joint work with Mark Stephenson and advised by Professor Saman Amarasinghe. They are as much the owners of this project as I am. Certain figures and tables in this document have been taken from papers coauthored with these collaborators. The experiments presented in Section 7.1 and Sections 8.1–8.3 were performed by Mark Stephenson and are included here to provide a complete presentation of the collaborative compilation project.

Mark has been an excellent colleague and a good friend, and he will likely be the best collaborator or co-worker I will ever have.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern aggressive optimizing compilers are very complicated systems. Each optimization pass transforms the code to perform better, but also affects the opportunities for optimization in subsequent optimization passes in ways that are difficult to understand. Each optimization pass relies on analyses of the code to determine the optimal code transformation, but these analyses may require an algorithm with unmanageable time complexity for an exact answer, and therefore can only be approximated by a heuristic. Online profiling can provide insight into the runtime behavior of a program, which aids the compiler in making decisions, but also adds complexity.

With a Just-In-Time (JIT) compiler, found in virtual machines for languages such as Java, the complexity grows. Because compilation happens online, the time spent compiling takes away resources from the running program, temporarily preventing the program from making progress. This means that optimizations must be fast, requiring greater use of heuristics. Although these heuristics give only an approximate solution, there are often parameters or settings that can be tuned to provide near-optimal performance for common programs.

These parameters are tuned by a compiler developer before the compiler is shipped. This "at-the-factory" tuning often optimizes for a suite of common programs, chosen based on the programs that users tend to compile. However, many programs do not lend themselves well to this process, because they do not have a consistent running time. Often programs that involve graphical user interfaces or network services are not conducive to benchmarking.

Moreover, the at-the-factory tuning can not possibly include the thousands of different programs that the compiler may be used to compile in the field. Although the compiler has been tuned for common programs, it may not do as well on a program that becomes popular after the compiler has been distributed. Many studies have illustrated the conventional wisdom that the compiler will not perform as well on benchmarks that are not in the at-the-factory training suite [42, 43, 10]. At an extreme, the compiler may be tuned to perform well on a small set of benchmarks, but the same performance can not be obtained with other typical programs [12, 31].

Collaborative compilation (also known as "altruistic compilation") allows users to use the programs they use every day as the benchmarking programs to tune their compiler, so that their programs perform as well as those in the benchmark suite. It builds on previous research on offline and online techniques for compiler tuning by creating a common framework in which to perform these techniques, and in addition allows users to transparently perform these techniques for the applications they use most. Because this is a rather costly endeavor for a single user, collaborative compilation allows users to divide the cost among a community of users by sharing information in a central knowledge base.

Although a virtual machine must take time on each run of an application to compile the application's code using the JIT compiler, collaborative compilation takes advantage of these repeated compilations to test different compiler parameters. In effect, it uses these compilations to help tune the compiler, just as we would at the factory. Because these compilations happen so frequently, collaborative compilation can collect just a small amount of data on each run, unlike many feedback-directed compilation techniques. The results of these tests are then incorporated into a central knowledge base. After gathering sufficient data from all community members, we can analyze the data to choose the best parameters. These parameters can be reincorporated into the users' JIT compiler to improve the performance on the applications that they use.

In the traditional compile/profile/feedback cycle, gathering data is an expensive proposition. These feedback-directed systems were designed to extract as much data as possible during the profiling phase. In contrast, a JIT is invoked every time a user runs an application on a VM. Thus, unlike a traditional feedback-directed system, a collaborative compiler does not have to gather data in wholesale. A collaborative compiler can afford to collect a small amount of information per invocation of the JIT.

Thus far, we have explained the general concept of collaborative compilation. Before describing how collaborative compilation works in more detail in Section 1.1, we give a short list of the founding principles for collaborative compilation:

Transparency Collaborative compilation must be transparent to the user. Once set up, the system should require no manual intervention from the user. In addition, the system must have negligible performance overhead, so that the user never notices a degradation in performance, only improvement.

Accuracy The information collected from community members must be accurate, regardless of system load during the time that the compiler parameters are tested. We must be able to compare and aggregate information from a diversity of computers and platforms.

Timeliness Collaborative compilation must give performance improvements quickly enough to satisfy users. Currently individual users can not perform the training required by "at-the-factory" methods because it is not timely.

Effectiveness Collaborative compilation must provide sufficient performance improvement to justify its adoption.

Extensibility We should be able to use collaborative compilation for any performance-related compiler problem. We must allow for a variety of techniques to tune the compiler, including profiling, machine learning, statistics, or gradient descent. These techniques will be described further in Chapter 2.

Privacy Collaborative compilation requires sharing information, potentially with untrusted individuals (possibly even publicly). Since the compiler has access to any data in the running application, ensuring that the user's data and activities remain private is important. The collaborative system should not sacrifice privacy to achieve the other attributes listed here.

Our implementation addresses all of these characteristics. Future work remains to adequately provide privacy and security.

1.1 Structure of Collaborative Compilation

Now that we have given a conceptual description of collaborative compilation, we will give a practical description. Collaborative compilation naturally fits into a client/server design, with each community member acting as a client and connecting to the server containing the central knowledge base. Clients test a compilation parameter by compiling programs with that parameter and measuring the performance of the compiled code. The client can then transmit the performance data to the central server, where it is collected and aggregated with data from other clients. The server then analyzes the aggregated data to arrive at a heuristic solution for the compiler problem. The server can publish this solution for clients to retrieve and use to improve the performance of their applications.

Collaborative compilation can tune a compiler for better performance based on the programs that the user frequently compiles and executes, as well as on characteristics of the user's machine. When compiling a program, the collaborative compiler compiles sections of the program in several ways, and inserts instrumentation instructions around these sections to measure the performance of the code. After gathering profile information during the program's execution, the system analyzes the data offline and reincorporates improved tuning information into the compiler. Since the tuning information was derived from the programs that were executed, those same programs will experience a performance boost.

The power of collaborative compilation is that data from many clients can be aggregated to provide better information than a single user could produce. However, this also complicates implementation, as data from diverse machines must be reconciled. We attempt to account for these variations or reduce their impact in a variety of ways, discussed in Chapter 4.

Because collaborative compilation is a very general framework, it can be used for many compiler problems, which can be tuned in many ways. Chapter 5 and Chapter 6 describe the compiler and tuning, respectively.

In the remainder of this thesis, we cover collaborative compilation in greater detail. In Chapter 2, we give background information on compiler optimizations, adaptive and profile-directed optimizations, and the machine learning techniques that we use in our system. Readers already familiar with this information may choose to skip parts or all of Chapter 2.

Chapters 3–6 give a detailed explanation of our implementation of collaborative compilation. Chapter 3 gives an overview of the implementation and defines terms used later on. Chapter 4 concerns measuring performance of a section of code, including problems that we have encountered and our solutions. Chapter 5 concerns collaborative compilation on the client. Chapter 6 concerns data transmission and the aggregation and analysis that the collaborative server performs.

Chapter 7 describes our experimental methodology, and Chapter 8 shows the results of these experiments. Chapter 9 describes recent work in the area of compiler tuning and its relationship to collaborative compilation. Finally, Chapter 10 concludes and gives directions for future work.

Chapter 2

Background

In this chapter, we give background information on several topics related to compilers and machine learning. We describe compiler optimizations, specifically method inlining and optimization levels. We discuss profile-directed optimization and how collaborative compilation fits into this category. We describe the adaptive system used by Jikes RVM, which makes online decisions on which optimization level to use. We then discuss how machine learning can be applied to compilation and two general techniques for doing so. In describing policy search, we introduce the priority functions that are used in many parts of our system, described in Chapter 5 and Chapter 6. Finally we explain supervised learning, and describe Ridge Regression, which we use in our experiments described in Section 7.2. Readers already familiar with this information may choose to skip parts or all of this chapter.

2.1 Compiler Optimization

This section describes inlining, a compiler optimization that is used in our experiments and extensively in examples, and describes optimization levels, which our experiments also targeted.

2.1.1 Inlining

Inlining is a fairly well-known optimization which removes the overhead of performing a method[1] call by replacing the call with the code of the method that is called. This can often improve performance in Java, which has many small methods. However, it can also degrade performance if it is performed too aggressively, by bloating code size with callees that are rarely called. Inlining can also affect compilation time, which, in a virtual machine, is a component of running time.

The benefits of inlining a function result not only from eliminating the overhead of establishing a call frame, but also because the caller becomes more analyzable. For example, many classes in the Java libraries are fully synchronized for multi-threaded use. However, this causes a great deal of unnecessary overhead when these objects are not shared among threads. Often after inlining the methods of such a class, the compiler can determine that the synchronization is unnecessary and remove this overhead. Similarly, inlining provides more options for the instruction scheduler to use the peak machine resources.

However, if the method becomes too large, performance may suffer due to register pressure and poor instruction scheduling.

Inlining has a non-monotonic unknown effect on compilation time. In virtual machines, only certain methods are compiled; which methods is unknown in advance (see Section 2.3). If a method X is inlined into a method Y, it will not need to be compiled separately, unless it is called by another method. However, if there is some other method Z that does call X, then X may be inlined into Z or compiled itself. In either case, X will effectively be compiled twice, once within Y, and again, either separately or within Z. As we will see in Section 8.4, compilation time is very roughly proportional to the number of instructions in a method. This means that if it takes t seconds to compile X separately, the compilation time of Y would increase by about t seconds when X is inlined, and similarly for Z.

[1]In the Java language, a function or procedure is called a "method." In our descriptions, we usually use "method" in the context of Java, and we use "function" in the context of mathematics. However, we may also use "function" to refer to a function in a general programming language.

2.1.2 Optimization Levels

Nearly all compilers can be directed to compile code at a given optimization level. The setting of the optimization level is an umbrella option that controls many specific settings and enables or disables sets of optimizations. When the user wants to generate high performance code, he or she can allow the compiler to take more time by specifying a higher optimization level. Likewise, if high performance is not necessary, a lower optimization level can be used. As examples, inlining is usually done at all levels, but is adjusted to be more aggressive at higher levels.

In a virtual machine, the choice of compilation level is critical; since additional time spent compiling takes time away from the running program, high optimization levels may not be the best choice. However, if a section of code is executed very frequently, it may be worth it to take extra time to compile it.

2.2 Profile-Directed Optimization

Profile-directed optimization improves the performance of the compiler by providing information about the actual behavior of the program, rather than relying on predictions based only on analyzing the code. Information can be collected about which sections of the program are executed, how often, in what context, and in what order. It may also gather information about the performance of the hardware, which can show which instruction cause cache misses or execution path mispredictions.

Although profile-directed optimization can be done separately, we describe it in the context of a virtual machine. As the virtual machine executes code, it may gather profile information that the built-in optimizing compiler can use to better optimize the code. The most common profile data gathered regards which methods are executed most frequently. The adaptive system uses this information to decide what to compile, when, and how. The implementation used in Jikes RVM is described in Section 2.3. In addition, ofter profile information indicating the frequently executed call sites is collected to aid in inlining decisions.

Most often, accurate and relevant profile information is not available until after the program has been compiled and run. Thus, to be truly useful, the profile information must be saved until the program is recompiled. In a Java virtual machine, recompilations happen frequently, since the program is recompiled when it is next executed. Recently, Arnold, Welc, and Rajan [5] have experimented with saving and analyzing the profile information to improve performance across runs. Collaborative compilation also saves and reuses profile information, but we are interested in the performance of a particular optimization. We relate the performance information to an abstract representation of the program, so that we can apply the profile information to previously-unseen programs, not only the programs which we have profiled.

2.3 Adaptive Compilation

As mentioned in Subsection 2.1.2, within a virtual machine, it is very important to choose the correct optimization level for a particular method. Choosing a level that is too low will result in slower code that will increase the total running time. However, choosing a level that is too high will result in time being wasted compiling, which also increases the total running time, if the method does not execute frequently.

This section describes the adaptive system used in Jikes RVM. This system uses a simple heuristic to decide when to recompile a frequently-executed or "hot" method [2]. The adaptive system uses a coarse timer to profile the execution of an application. When a method accrues more than a predefined number of samples, the method is determined to be hot. When a method is deemed hot, the adaptive system evaluates the costs and benefits of recompiling the method at a higher level of optimization. The adaptive system simply ascertains whether the cost of compiling the method at a higher level is worthwhile given the expected speedup and the expected amount of future runtime. It chooses the optimization level l^* with the lowest expected cost:

$$l^* = \operatorname*{argmin}_{l \geq k} \{C_l + \frac{P}{B_{k \to l}}\}$$

Here l and k are optimization levels and k is the current optimization level, C_i is the cost of compilation for the method at level i, P is the projected future remaining time for the method, and $B_{k \to l}$ is the benefit (speedup) attained from recompiling at level l when the current compiled method was compiled at level k. $C_i = 0$ if the method is already compiled at level i. Likewise $B_{i \to i} = 1$.

While the heuristic is intuitive and works well in general, it makes the following simplifying assumptions:

- A method's remaining runtime will be equal to its current runtime.
- A given optimization level i has a fixed compilation rate, which is used to compute C_i, regardless of the method the VM is compiling.
- Speedups from lower to higher levels of optimizations are independent of the method under consideration (*i.e.*, the B's are fixed). For instance, it assumes that optimization level 0 (`O0`) is 4.56 times faster than the baseline compiler (`O-1`), regardless of which method it is compiling.

Several of our experiments, described in Chapter 7, try to improve on these assumptions.

If the adaptive system determines that the benefits of compiling at a higher level (improved running time) outweigh the costs of compilation, it will schedule the method for recompilation. The time at which this happens is determined solely by the hot method sampling mechanism, and is nondeterministic. Thus, the adaptive system could replace a method with a newly-compiled version at any time.

2.4 Applying Machine Learning to Compiler Optimization

As mentioned in Section 2.2, collaborative compilation collects profile information from program executions, and applies this information when compiling any program, not only the program previously profiled. This section describes how this is possible.

Compiler optimizations often use a heuristic to make decisions about how to optimize. These heuristics are often based on metrics or characteristics of the region of code being compiled, which we call features. The set of features used depends on the task at hand, as does the granularity. In this project, we collected features at the granularity of a single method.

Compiler developers often create heuristics that take as input a vector of features for a particular program, method, loop, *etc.* and produce as output a compilation decision. Previous research has shows that these heuristics can successfully be created automatically. We describe two processes that can be used to create the heuristics automatically in Section 2.6 and Section 2.5.

With the features and the heuristic function, we can specifically direct the compiler in compiling a method. We extract the features from the method, apply the function to get a compilation decision, and compile using this decision. We can measure the worth of the heuristic by the performance of the code that it produces.

2.5 Policy Search and Priority Functions

In [43], Stephenson *et al.* used a form of policy search to learn the best way to apply hyperblock formation, register allocation, and data prefetching. They were able to extract the decisions that the optimization makes into a single priority function. Using these priority functions, they were able to use genetic programming, a type of policy search, to improve the performance of these optimizations.

In general, policy search is an iterative process that uses a directed search to find the best policy for a problem. The policy simply describes what decision should be made in each situation, where the situation is described by a feature vector. The policies are rated by how well they do on a particular problem.

There are many ways to search for the best policy; a popular choice is genetic programming, which is loosely analogous to biological evolution. With genetic programming, an initial population of policies are created, perhaps randomly. Each of these policies are rated to determine their value. The most valuable policies are kept for the next generation. Some percentage of these may be mutated to create slightly different policies. Others may be combined in some fashion to create a new policy

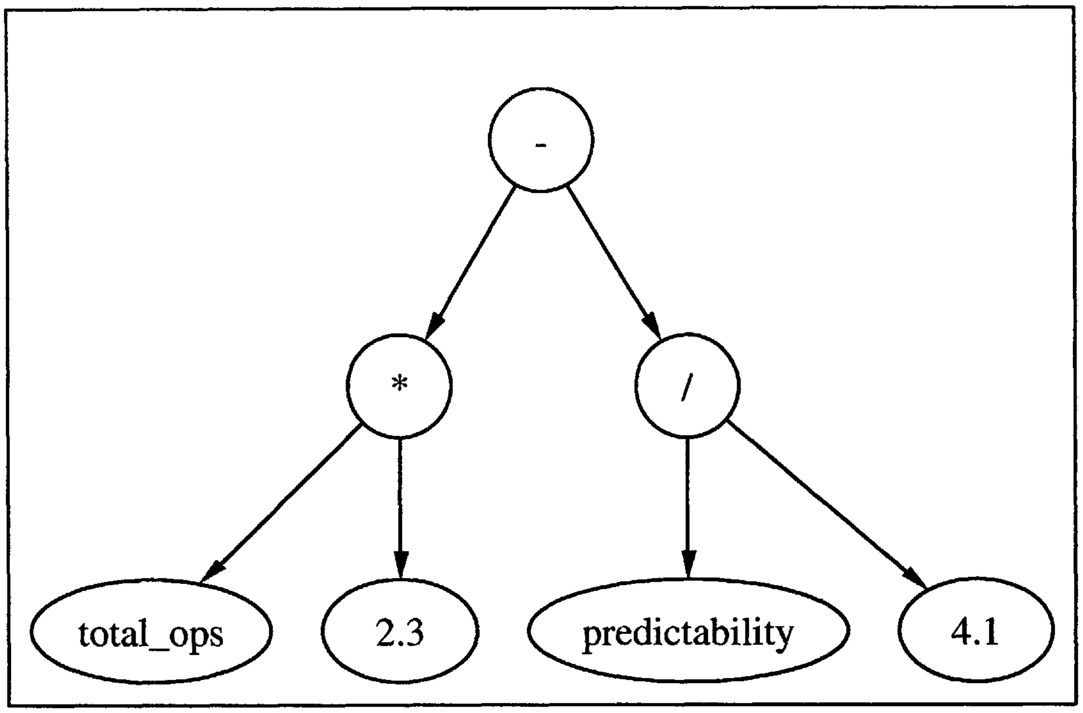


Figure 2-1: Example Priority Function. This illustrates the form of a priority function, taken from Stephenson *et al.* [43]. The priority function can be mutated to create a similar function.

with characteristics of both. With this new population, the policies are again rated, and the process continues.

To give an example, Stephenson *et al.* [43] used a simple expression tree as a policy. The expression tree has terminals that are numbers, booleans, or variables. A number of subexpressions can be combined using arithmetic, comparison, and logic operators. This forms an expression tree, as illustrated in Figure 2-1. Compiler developers are familiar with this representation of functions, as it is akin to an Abstract Syntax Tree, the result of parsing source code.

The priority functions can be randomly created by choosing random operators and terminals, up to a certain depth. They can be mutated by replacing a random node with a randomly-created subexpression. Two priority functions can be combined using crossover; a subexpression from one function is inserted at a random place in the second expression.

By using genetic programming with these policy functions, Stephenson *et al.* were able to successfully improve the performance of compiler optimizations.

2.6 Supervised Learning

Supervised learning is similar to policy search in that the goal is to arrive at a function that makes the correct decisions in each situation. However, with supervised learning, more information is available. We are given a set of feature vectors which describe some situations, and also the correct decision for each feature vector. The correct decision, or label, for each feature vector is provided by a "supervisor." So, if we have a small set of feature vectors, each of which is labeled (the training set), we can use supervised learning to extrapolate from this small set to produce predictions for feature vectors that we have not seen (the test set).

Stephenson and Amarasinghe [42] have also applied supervised learning to compilation problems. This works in a way similar to that of policy search. We simply need to find the best compilation decision to make in a set of situations, then use machine learning to extrapolate from this set. However, the training set must be large enough to capture the factors involved in the decision. As the training set grows larger, the predictions improve.

In Section 7.2 we will describe an experiment using the collaborative system to improve the Jikes RVM adaptive system by using supervised learning to make more accurate compilation time predictions. The learning technique we use is Ridge Regression, a type of linear regression that employs coefficient shrinkage to select the best features. We summarize a description of Ridge Regression from Hastie, Tibshirani, and Friedman [20]. Linear regression produces the linear function of the input variables that most closely predicts the output variable. Specifically, linear regression produces the linear function $y = \beta \cdot \mathbf{x} + \beta_0$ such that the sum of squared errors on the training data set is minimized:

$$\beta, \beta_0 = \underset{\beta,\beta_0}{\operatorname{argmin}} \sum_i \left(y_i - \beta \cdot \mathbf{x}_i - \beta_0\right)^2 .$$

Ridge regression is an extension of standard linear regression that also performs feature selection. Feature selection eliminates input features that are redundant or only weakly correlated with the output variable. This avoids over-fitting the training data and helps the function generalize when applied to unseen data. Ridge regression eliminates features by shrinking their coefficients to zero. This is done by adding a penalty on the sum of squared weights. The result is

$$\beta, \beta_0 = \operatorname*{argmin}_{\beta,\beta_0} \left\{ \sum_i (y_i - \beta \cdot \mathbf{x}_i - \beta_0)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

where p is the number of features and λ is a parameter that can be tuned using cross-validation.

2.7 Summary

These techniques have allowed compiler developers to collect performance data from specific benchmarks and train heuristics at the factory. However, Chapter 1 points out the problems with this approach. Collaborative compilation uses profile-directed optimizations to bring the benefits of at-the-factory training to individual users.

Chapter 3

Implementation Overview

This chapter gives a general overview of our implementation of collaborative compilation. First we introduce some terminology, then describe the system using this terminology. As we mention each topic, we point out where in the next three chapters the topic is explained in detail. We finish the chapter with implementation details for the system as a whole.

3.1 Terminology

The goal of collaborative compilation is to improve a heuristic for a particular compilation problem. The collaborative system provides a very general framework that can be used for many compilation problems and for many types of profile-directed and feedback-directed optimizations as well as offline machine learning and artificial intelligence techniques. The collaborative system attempts to unify these varied techniques so that very few components are specific to the employment of the system. Our description necessarily uses ill-defined terms to describe the components that are dependent on the particular use of the system. Here we attempt to give a definition of three general terms.

Optimization The component of the compiler that we want to tune. We use the term "optimization" because an optimization pass is a natural component to tune, but other components can also be tuned—in fact, our experiments mainly target the adaptive system of Jikes RVM. The optimization usually must be modified to provide optimization-specific information to the collaborative system and to make compilation decisions based on directives from the collaborative system.

Analyzer This component abstracts the details of how performance data is analyzed and used to tune the compiler. As mentioned, the system could use many techniques, which may need to be customized for the optimization being tuned. This component is described in more detail in Section 6.2.

Compilation Policy A compilation policy is used in many ways, but at the basic level, it is simply a means of communication between the optimization and the analyzer. It is through the compilation policies that the collaborative system directs the optimizations. A compilation policy also acts as the "best answer" for tuning the optimization. Compilation policies are defined and explained in Section 5.2.

3.2 Interaction of Components

Using these three terms, we can explain the behavior of the collaborative system. First, we set up a collaborative server (Section 6.5) using the analyzer (Section 6.2). Then, a client connects to the server, and retrieves one or more compilation policies to test (Subsection 6.5.2). The client then uses these compilation policies to compile a small part of the application, chosen randomly (Section 5.1), and runs the application, occasionally gathering performance information about this code section (Chapter 4). The client then sends the performance information and any optimization-specific information to the server (Subsection 6.5.1).

After a sufficient amount of data has been collected (Section 6.1), the analyzer can analyze the data to produce a compilation policy that best tunes the optimization (Section 6.2). The data may also narrow the compilation policies that should be tested. The client can retrieve the best compilation policy and use it to improve

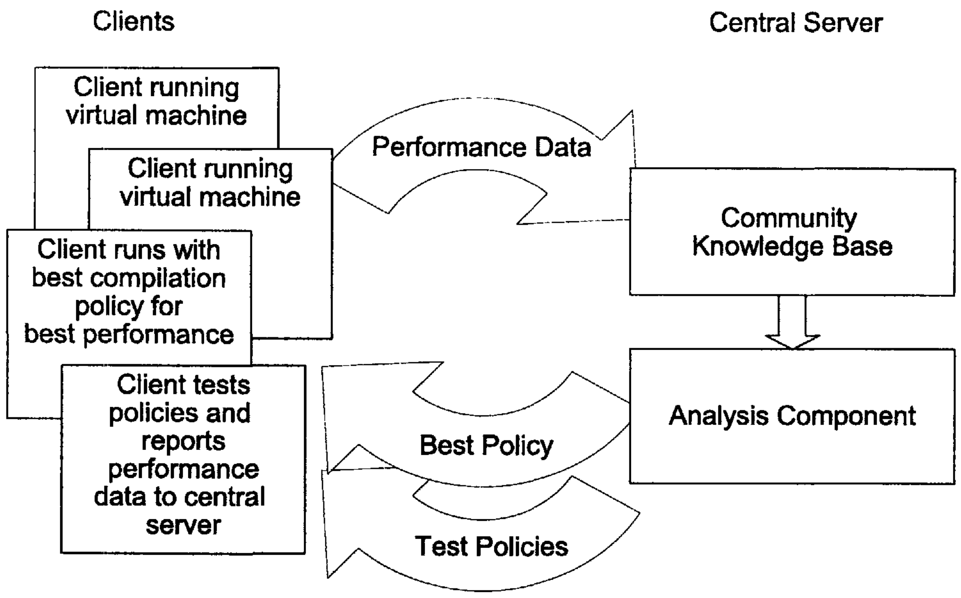

Clients
Central Server
Client running virtual machine
Client running virtual machine
Client runs with best compilation policy for best performance
Client tests policies and reports performance data to central server
Performance Data
Community Knowledge Base
Analysis Component
Best Policy
Test Policies

Figure 3-1: Flowchart of Collaborative Compilation. This diagram illustrates how the components fit together.

performance, and continue to retrieve and test other compilation policies (Section 5.3). An illustration of this feedback process is given in Figure 3-1.

Throughout this description, we present the system as improving a single optimization. However, it is simple to extend the system to tune several optimizations at once. In many cases this involves simply tagging data with the related optimization. Other components can be replicated and customized for each optimization.

3.3 System Implementation Details

As mentioned in Section 1.1, we implemented collaborative compilation in Jikes RVM, a Java virtual machine that performs no interpretation, but rather takes a compile-only approach to execute the compiled Java bytecodes. Our modifications to Jikes

RVM included the following:

- Adding an optimization pass that adds collaborative instrumentation to selected methods, performed in the HIR (high-level intermediate representation). This pass is described in more detail in Section 5.1.
- Adding several global fields that can easily be modified by compiled code. This included a reference to an object in which the measurement data is stored, and a reference to the thread that is running instrumentation code.
- Modifying the thread scheduler to allow the system to detect thread switches, as described in Subsection 5.4. Jikes RVM performs a thread switch in order to perform garbage collection, so this technique also detects garbage collections.
- Modifying the exception handler to detect exceptions. If an exception occurs while measuring, we discard the measurement.
- Modifying optimization passes and the adaptive system to be controlled using the collaborative system, described in Chapter 5.
- Adding the ability to communicate with the collaborative server, described in Section 6.5.
- Adding command line options to direct collaborative compilation.
- Adding various classes to help with the above tasks.

We also implemented the collaborative server in Java, described in Chapter 6. All analyzers and compilation policies described have also been implemented in Java. We made use of the Junit unit test framework [24] and the Jakarta Commons Mathematics library [13].

The next three chapters describe three main concerns of the collaborative system in detail.

Chapter 4

Measurement

Gathering accurate information is an important requirement for collaborative compilation. We must be able to collect reliable information despite the wide diversity of environments in which the system could be used. This chapter describes how we account for differences between systems and noisy systems to measure performance accurately.

In our experiments, we chose to measure performance by the time the code takes to execute, although other resources, such as memory usage or power consumption, could be measured if desired. Indicators of processor performance could also be used to measure performance, such as cache misses, branch mispredictions, or instructions executed per cycle.

4.1 Sources of Noise

There are countless sources of measurement error that must be eliminated or accounted for by collaborative compilation. In this section we will discuss the sources of noise that we are aware of that cause the greatest problems; in Section 4.2 and Section 4.3 below, we will discuss the techniques that we used to overcome these problems.

There are three general sources of measurement error in the collaborative system, which we will explain in more detail throughout this section. First, measurements made in different environments can not be directly compared. Second, transient effects prevent us from comparing measurements made at different times. Third, our measurements can be disrupted, giving us incorrect results.

Several sources of noise come from the Java virtual machine in which we have implemented collaborative compilation. The virtual machine is a complex system with quite a few interacting components: the core virtual machine, a garbage collector, an optimizing compiler, an adaptive compilation system, and a thread scheduler. All of these components cause problems for measurement, either alone (the garbage collector, the adaptive system, and the thread scheduler) or in combination with other effects (the optimizing compiler and the virtual machine).

4.1.1 Machine Type and Environment

Collaborative compilation aggregates data from many users that may be running very different hardware. Clearly we can not directly compare timing measurements recorded on different machines. Moreover, even on identical systems, other environmental effects, such as system load or contention for input and output devices, can affect the measurements. We must account for the diversity of computers and software environments.

4.1.2 Contextual Noise

The performance of a section of code can be influenced by the context in which it runs. This effect can occur in several ways. First, the CPU maintains state in the data cache, instruction cache, and branch predictor, which depends on the sequence of instructions preceding the measurement. If there are several contexts in which the code executes, for example calls to the same method in different parts of a program, the processor will be in different states in each context. In addition, after a thread switch or context switch, the state is quasi-random. Second, a method called in different places in the program is more likely to have consistently different inputs, due to the multiple contexts in which it is called.

4.1.3 Program Phase Shift

Programs typically have several phases, as described in [40]. A simple pattern is reading input in one phase, performing a computation in another phase, and outputting the result in a third phase. Programs can have many more phases, and alternate between them.

As it regards collaborative instrumentation, program phases may be a problem if an instrumented code region is executed in more than one phase. This is problematic because the calling context has changed. We described above the effect that this may have on the CPU. Different calling contexts often indicate different inputs as well.

A new phase may cause the virtual machine class loader to dynamically load new classes. This can invalidate certain speculative optimizations that assume that the class hierarchy will remain constant, which will cause performance to drop suddenly. The new types may be used where a different type had been used before, which changes the behavior of the code.

4.1.4 Adaptive System

The virtual machine's adaptive system could replace a method with a newly-compiled version at any time (see Section 2.3). If a code segment that we are measuring contains a call to the updated method, the measurements will suddenly decrease.

4.1.5 Garbage Collection

The virtual machine provides a garbage collector to manage the program's memory. There are two common ways to allocate memory in a garbage collected region of memory [23]. With a copying/bump-pointer allocator, live objects are copied to make a contiguous section of live objects. This leaves the rest of the region empty for allocation, and the space can be allocated simply by incrementing a pointer. With a sweeping/free-list allocator, objects found to be garbage are reclaimed and added to a list of unoccupied spaces, called the free-list. When a new object is allocated, the free-list is searched until finding a space large enough for the object.

Both of these allocation schemes cause problems with measurement. If a copying collector is used in the data region of memory, objects will change locations after a garbage collection. This may have a significant impact on data locality and transiently impacts the cache performance, since objects may be moved to different cache lines. Therefore, the performance of the code may change.

If a free-list allocator is used in the data region of memory, then the time required to allocate objects is not consistent, especially for larger objects. If the code being measured contains an object allocation, the measurements may not be consistent. If it is not important for the measurements to include the allocation time, we could insert additional instructions to stop the timer before allocating an object and restart afterward. However, any optimization that affects allocation time (for example, stack allocation) will require that we reliably measure this time. For generality, our system should work with these optimizations.

With any type of garbage collector, when the garbage collector runs, it will evict the data in the data cache and likely the instruction cache as well. Our implementation detects garbage collections so that they do not affect the measurements.

4.1.6 Context Switches

By their nature, operating system context switches can not be detected by the application. If a context switch occurs while measuring, the time for the context switch will be included in the measurement.

Although thread switches could be as problematic as context switches, the virtual machine is responsible for managing its threads, so we can detect when a thread switch has invalidated our measurements. We give more details on this in Subsection 5.4.

4.1.7 Memory Alignment

We observed that occasionally two identical methods placed in different locations in memory differed substantially in performance, as shown in Figure 4-1. This agrees with the findings of Jiménez, who attributes this effect to the conditional branches

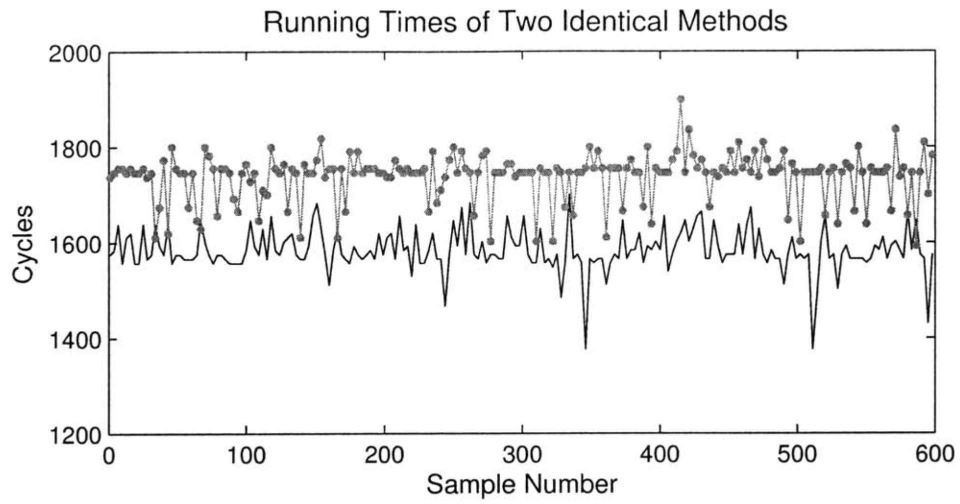


Figure 4-1: Effect of Differing Memory Alignment on Performance. This plot shows the measured running times of two identical methods. The two methods consistently differ in performance by about 10%. (The measurements have been median filtered using the technique in Subsection 4.2.5. The x-axis of the plot is scaled to correspond to the original sample number.)

having different entries in the branch history table [22]. Simply by inserting no-ops (instructions which do nothing), Jiménez was able to affect performance as much as 16% by reducing conflicts in the branch history table.

4.2 Measurement of Running Time

We have implemented a measurement technique that successfully eliminates or accounts for a great deal of the noise described in Section 4.1. First we explain the basic implementation of an instrumented section of code, and then progressively extend this basic model to reduce noise to a low level. In Chapter 8 we show experimental results that validate our approach.

4.2.1 Basic Measurement

To instrument a particular section of code to obtain its running time, we time it by adding instructions to start a timer before the code executes and stop the timer afterward. We run the instrumented code many times to collect measurements, then compute the mean and variance. We compute these values incrementally by accumulating a running sum, sum of squares, and count, of the measurements. If $\sum_i x_i$ is the sum, $\sum_i x_i^2$ is the sum of squares, and n is the count, we use the standard formulas for sample mean, $\bar{x}$, and sample variance,[1] s_x^2:

$$\bar{x} = \frac{1}{n}\sum_i x_i$$

$$s_x^2 = \frac{1}{n(n-1)}\left[\left(n\sum_i x_i^2\right) - \left(\sum_i x_i\right)^2\right]$$

We allow several sections of code to be measured simultaneously by keeping the sums, sums of squares, and counts for each section in a distinct container object.[2] In the text that follows, we will be extending this model to include further information. However, since this is not a mystery novel, we present the final result here in Figure 4-2, and point to the section where each field is explained. The reader may want to refer back to Figure 4-2.

A very important goal of collaborative compilation is to be transparent to the end user, which includes being as unobtrusive as possible. Thus our instrumentation can add only a slight performance overhead. We limit the overhead by taking samples of the execution times at infrequent intervals, similar to the approach of Liblit *et al.* [27] and Arnold *et al.* [4]. So, to instrument a section of code, we add instructions at the beginning of the section to randomly choose whether to take a sample or to execute the unmodified code. The probability of choosing to sample is small. The instrumented path is a separate copy of the original code, so that the only overhead for the regular path is the random choice.

[1]An anonymous reviewer recently informed us that this formula for sample variance suffers from numerical instability. He or she suggested an alternative method, which is presented as West's Algorithm in [11]. We have not yet implemented this method.

[2]For simplicity, we use the container object to explain our implementation. Our actual implementation stores this data in a single, global array for all code sections.

```
class MethodData {
    FeatureVector features;              // Discussed in Section 5.2
    CompilationPolicy version_1_policy; // Discussed in Section 5.2
    CompilationPolicy version_2_policy; // Discussed in Section 5.2
    MeasurementData version_1_copy_1; // Discussed in Subsections 4.2.2,
    MeasurementData version_1_copy_2; //    4.2.3, and 4.2.6
    MeasurementData version_2_copy_1; //
    MeasurementData version_2_copy_2;   //
}
class MeasurementData {
    long sum;                            // Discussed in Subsection 4.2.1
    long sum_of_squares;                  //
    long count;                          //
    long temporary_1;                   // Discussed in Subsection 4.2.5
    long temporary_2;                     //
    long compilation_time;                // Discussed in Section 4.3
}
```

Figure 4-2: Data Storage Objects. This illustrates the information that is kept for each method that is collaboratively compiled. The sections in which each field will be defined are given in comments at the right. The two copies of two versions are a result of combining the techniques in Subsections 4.2.2, 4.2.3, and 4.2.6

4.2.2 Relative Measurement

We can account for much of the noise described previously by comparing two versions of the code in the same run of the program. These two versions perform the same function, but are compiled in different ways. We then sample each version multiple times in a program run. Since the two versions are measured near the same time, the machine type, environment, and program phase are approximately the same for both versions. If the adaptive system updates a callee, as described in Section 4.1.4, the two versions see the same update.

4.2.3 Granularity of Measurement

We measure at the granularity of a single method invocation. That is, we start a timer before a single method call and stop the timer after the call. This allows for a large number of samples with minimal impact from the timing code. Measuring at the method level makes the measurements short enough so that relative measurement works well. We considered measuring at other levels of granularity, but the method level seemed most appropriate. With a broader level, such as measuring a run of a program, fewer samples would be collected, so differences in time due to changing inputs and a changing environment become important. With a finer level, such as measuring a basic block, the code used to measure the performance affects the measurement itself.

4.2.4 Random Sampling

In Subsection 4.2.1 we explained that the decision of whether or not to sample a method is a random choice for each invocation. If the method is sampled, the choice of which version to sample is also random. This ensures that the contextual noise, including the inputs to the method, is distributed in the same way for each version. Random sampling, along with redundant copies, explained below, tends to eliminate the effect of contextual noise. For memory allocation, discussed in Subsection 4.1.5, random sampling also helps to ensure that the average of several measurements containing allocations will be near the average-case time for allocation.

4.2.5 Median Filtering

As mentioned in Subsection 4.1.6, context switches can not be detected by the collaborative system. Thus, our measurements sometimes include the time for a context switch, which are often orders of magnitude larger than the true measurements. In order to remove these outliers, we use a median filter on the measurements. For every three measurements, we keep and accumulate only the median value.

To implement median filtering, for each version of each method, we reserve slots for two temporary values, as shown in Figure 4-2. The first two of every three measurements are recorded in these slots. Upon collecting a third measurement, the median of the three numbers is computed and accumulated into the sum, with the count and sum of squares updated accordingly. Using this approach, measurements that are orders of magnitude larger than the norm are ignored.

Just as for context switches, it is possible for the virtual machine thread scheduler to trigger a thread switch while we are measuring. We can detect thread switches and therefore prevent incorrect measurements. This is discussed in Subsection 5.4.

4.2.6 Redundant Copies

As mentioned in Subsection 4.1.7, memory alignment of the compiled code can have significant impact on performance. Since the two versions of the method are at different memory locations, there is a potential for the two versions to perform differently due to alignment effects, rather than due to the quality of compilation.

To reduce the likelihood of this effect and other types of noise, we developed a way of detecting when the memory alignment has a significant impact on performance. We accomplished this by using redundant copies of each version of the method that we are measuring. We collect data for each redundant copy in the same way that we have described thus far for a single copy. This is illustrated in Figure 4-2 as the two copies of each version. When we process the measurement data, we compare the two redundant copies. If they differ significantly, we throw out these measurements. Otherwise, we can combine the sums, sums of squares, and counts of the two redundant copies to get a better estimate of the performance. Our implementation requires that the means of the measurements for the two copies differ by at most 5%.[3]

Although this does not guarantee that noise due to memory alignment will be eliminated, our experimental results in Section 8.1 show that this method effectively reduces a great deal of noise.

4.3 Measurement of Compilation Time

So far, we have discussed only runtime measurement. However, in a Java virtual machine, compile time is a component of runtime, so there is a trade-off between compiling more aggressively for a shorter runtime or compiling less aggressively, but quickly. This trade-off is discussed in Section 2.3. This section describes how we collect compilation time data so that we can account for this trade-off.

As a byproduct of our implementation of redundant methods, described in Subsection 4.2.6, we compile each version of a method twice. This gives us two compile time measurements for each run. We use wall-clock time to measure the compilation time, which is usually accurate enough for our purposes.

4.3.1 Relative Measurement

When measuring running times, we were able to collect many measurements in a single run and aggregate them to get a better estimate of the expected time. However, performing many extra compiles would create an unacceptable overhead for the user. Thus, we need to be able to compare compile times across runs and across machines.

The designers of Jikes RVM found a solution to this problem [2]. In order to use the same tuning parameters for any computer, Jikes RVM compares the compile times of the optimizing compiler to the "baseline" compiler. The assumption is that

[3]We also implemented an unpaired two-sample t-test at significance level 0.05 to decide if the measurements for the two copies have equal means, but this test was not used in any of the experiments presented in Chapter 7.

the processor speed and other effects affect the compile times of the baseline compiler and optimizing compiler by the same factor. Thus, the ratio between the optimizing compiler and baseline compiler will be the same on any system. By measuring compile time in units of the baseline compiler, we can successfully compare the ratios across runs and across systems.

4.3.2 Outlier Elimination

Like the running time measurements, the measurements for compile time could include context switches. Rather than attempt to detect context switches on the client, we remedy this problem when the measurements have been collected by the collaborative server. The details of aggregating and sorting the measurements is described in Section 6.1. Once we have a set of measurements for a single method's compilation time, we can detect those measurements that may include a context switch because they are significantly larger than the other measurements. We drop the highest 10% of the measurements to remove these outliers. In order to avoid biasing the mean toward smaller numbers, we also drop the lowest 10% of the measurements.[4]

4.4 Measurement Assumptions

This section summarizes some of the assumptions we have made in our measurement system. Future work remains to reduce these assumptions.

4.4.1 Large Sample Size

We assume that we will have an abundance of running time and compilation time measurements, so that we have the liberty of throwing out measurements that seem to be incorrect and so that we can rely on the law of large numbers [39] to get a good estimate for the time even though the measurements are noisy. If enough users

[4]This procedure effectively brings the mean closer to the median. One might consider simply using the median value instead.

subscribe to collaborative compilation, this amount is achievable without causing any single user an unacceptable level of overhead.

4.4.2 Similar Environment

We assume that the environmental noise, including the machine environment and contextual noise, is approximately the same for each version. Because we randomly choose the version to measure, this should be true on average if enough samples are collected. This assumption is required by any method of measuring, since it is not possible to measure two methods under exactly the same environment. However, with collaborative compilation, we have less control over the environment.

4.4.3 Similar Contexts

We assume that each version and redundant copy of a method will have approximately equivalent distributions over the contexts in which the method is called, and that this distribution is the same as the unmodified method. We require this assumption to account for differing inputs. Again, this should be true on average if enough samples are collected.

4.4.4 Relatively Short Methods

We assume that the execution time of a method is small compared to program phase shifts or changes in the environment. Otherwise, the assumption of similar environments can not hold.

4.4.5 Baseline Compilation Time

We assume that the speed of the optimizing compiler is a constant factor of the speed of the baseline compiler for all systems. We have not tested this assumption empirically, although it is assumed in [2], and it seems to work well in practice.

Chapter 5

Compilation

The last chapter described how to measure the running time and compilation time for a method. It explained that the collaborative system compares the performance of two versions of a single method. (The fact that the system also compares two copies of each version will not be relevant until Section 5.1.) This chapter describes how each version of the method is compiled, and how the system chooses the version to create. Section 5.2 describes how the compiler is directed differently for each version, and Section 5.3 describes how a collaborative server can request specific information in order to coordinate collaborative clients. Section 5.1 follows up with details on how the methods are instrumented and sampled.

5.1 Compiling and Instrumenting Methods

As the optimizing compiler compiles methods, our system randomly chooses one of these methods to collaboratively compile. It then inserts a section of code at the beginning of the method that will occasionally take a measurement of one collaboratively-compiled version. The remainder of the method is untouched.

The system then schedules[1] additional compilations of the method, using two different compilation policies. This creates two versions of the method that we will

[1]Jikes RVM provides a convenient way of scheduling specialized methods that we have used for this purpose.

compare.

In addition, the system uses redundant copies of the same version of a method to eliminate certain types of noise, as described in Subsection 4.2.6. These redundant copies are also scheduled for compilation, using the same compilation policies as the two existing versions.

When each of these four methods (two versions with two copies each) is compiled, we record the compilation time along with the other data for the method. This is illustrated in Figure 5-1.

As mentioned above, the system inserts control flow into the original method to decide whether or not to take a sample of one of the collaboratively-compiled versions. As described in Section 4.2.1, this code flips a biased coin to decide if the method will be sampled. If not, control falls through to the original version of the method. If so, another unbiased coin flip determines which of the four alternative methods to execute. Each of the four method calls are embraced by instrumentation code that times the method invocation and records the elapsed time, after which the result of the method call is returned as the result of the original method.

5.2 Compilation Policies

As described in Section 2.5 and Section 2.4, a policy is a function from features to an action. We use compilation policies in the collaborative compiler to take features that describe a method abstractly, and produce a compilation decision, which directs the actions of the compiler. Since this is a very abstract definition, we will give examples below to illustrate.

The features may include the results of analyses of the method, such as liveness analysis or escape analysis; or statistics about the method, such as number of branches, number of calls, or number of instructions that could potentially run in parallel. Compiler developers use features such as these in manually-created heuristics that make compilation decisions for an optimization. We will describe the details of how features are collected and used in creating a collaboratively-generated heuristic function.

The collaborative system uses a compilation policy to direct the compiler when compiling each alternative version of the method, and are optimization-specific. Compilation policies include policy functions that take features as input and produce a compilation decision as output. A compilation decision is a specific value, parameter, or option that directs a particular aspect of the compilation of a method. A compilation decision can also act as a compilation policy; it is like a function with no inputs.

The simplest compilation policy is a boolean indicating whether or not to run a certain optimization pass. A compilation policy might also be a parameter of an optimization pass. The compilation policy may also be more complicated. For example, a compilation policy for inlining may need to indicate a particular call site that may not even be in the method being compiled, and may not be known in advance. For policy search, in most cases the compilation policy is the policy itself. For supervised learning, the compilation policy must not be a policy function.

When each version of a method is compiled by the collaborative system, the necessary features are collected. If the compilation policy is a policy function, the features are used as input to this function to get the compilation decision that should be used. The policy function could be used many times during a single compilation, as was the case for the priority functions used in [43], described in Subsection 2.5.

However, if the compilation policy is a compilation decision, the features are stored and associated with the performance data for the method. The goal of collecting this information is so that a heuristic function can be collaboratively-generated. In order to create such a function, the collaborative system will need to relate the features, the compilation decision used, and the resulting performance of the code. Once we have compiled each copy of each version and collected enough samples from the method to determine performance, the features are sent to the collaborative server along with the compilation decision for each version and performance data for each version. Figure 5-1 has been copied from Figure 4-2, and illustrates the information that is kept for each method and sent to the collaborative server.

```
class MethodData {
    FeatureVector features;               // Discussed in Section 5.2
    CompilationPolicy version_1_policy; // Discussed in Section 5.2
    CompilationPolicy version_2_policy; // Discussed in Section 5.2
    MeasurementData version_1_copy_1; // Discussed in Subsections 4.2.2,
    MeasurementData version_1_copy_2; //    4.2.3, and 4.2.6
    MeasurementData version_2_copy_1; //
    MeasurementData version_2_copy_2;  //
}
class MeasurementData {
    long sum;                            // Discussed in Subsection 4.2.1
    long sum_of_squares;                 //
    long count;                          //
    long temporary_1;                    // Discussed in Subsection 4.2.5
    long temporary_2;                    //
    long compilation_time;               // Discussed in Section 4.3
}
```

Figure 5-1: Data Storage Objects. This figure has been copied from Chapter 4 as a convenience. The sections in which each field is defined are given in comments at the right. In this chapter, the fields are defined in this section, Section 5.2.

5.3 Heuristics from Collaborative Server

With supervised learning, the set of valid compilation policies that can be used to direct compilation is usually known by the virtual machine, and one can be chosen at random for the compilation of each version. However, if there are a large number of possible compilation policies, or if the compilation policy is a function, then the client will need to contact a collaborative server to retrieve the compilation policies that it should use. In this way, the server can coordinate the collaborative clients so that the server can gather enough data points for a specific compilation policy to judge the policy's value.

The collaborative system includes a way to transfer compilation policies in clear text from the server to the client. The client can parse and use the compilation policies to compile the two versions of each method that is collaboratively instrumented. In addition, since the client has the ability to download functions, the collaborative server can also publish the current best-performing heuristic function, which the client can use to improve the performance of all the code that is optimized. The details of the system that transfers compilation policies is described in Section 6.5.2.

5.4 Implementation Details

In this section we explain several details of initialization and recording. These details show how we correctly measure as described in Chapter 4. We show that the system randomly samples with low probability and samples each version with equal probability. We also show how we correctly measure the method invocation and perform median filtering. Additionally, we describe our method of detecting thread switches in detail.

Assume that there are global variables `collaborativeSamplingThread` and `nextSampleTime`, and methods `getCurrentThread()` that returns a reference to the current thread, and `findMedian(long x, long y, long z)` that finds the median of the three arguments.

The instrumentation code updates a global variable to indicate the thread that is currently measuring. We allow only one thread to measure at a time. Upon starting the timer, the global variable is set to the current thread, and upon stopping the timer, this variable is checked to ensure that it has not changed before accumulating the measurement and setting the variable to `null`. When a thread switch occurs, this global variable is also set to `null`, so that if there is a thread measuring, the measurement is discarded.

When choosing a version to sample:

1. `collaborativeSamplingThread` is checked. If it is not `null`, then another method is currently being sampled, and we do not sample this method.

2. We check if we should sample. Using the technique of Liblit *et al.* in [27], we simply decrement `nextSampleTime` and compare it with zero. If we should not sample, execution continues with the original method code.

3. Since `nextSampleTime` has reached zero, we reset it to a random value in a

range based on our sampling frequency.

4. We set `collaborativeSamplingThread` to `getCurrentThread()`.

5. We generate a random number between 1 and 4, inclusive, and choose the instrumented method based on this value.

6. We save the current time (using the processor cycle count) and call the chosen method.

After the instrumented method has finished:

1. We get the current time and subtract the starting time to get a measurement of the method execution time, **`newMeasurement`**.

2. We check that `collaborativeSamplingThread` is equal to `getCurrentThread()`. If a thread switch occurs, `collaborativeSamplingThread` is reset, so effectively this checks if a thread switch has occurred while measuring this method. If not, we discard `newMeasurement` and continue with step 6. Otherwise we reset `collaborativeSamplingThread` to `null`.

3. We get the `MeasurementData` object associated with this method.[2] The fields of the `MeasurementData` class are shown in Figure 5-1.

4. If `count` $= 0 \pmod 3$, we compute `findMedian(temporary_1, temporary_2, newMeasurement)`. We add the resulting value to `sum`, square it and add it to `sum_of_squares`, and increment `count`.

5. Otherwise, we save `newMeasurement` in `temporary_1` or `temporary_2`, depending on the value of `count` $\bmod 3$. We increment `count`.

6. Finally, we return from the method, using the return value of the instrumented method (or no return value, if appropriate).

[2]As mentioned in Section 4.2.1, we store all measurement data in a single array. We explain using the `MeasurementData` objects for clarity. A `MeasurementData` object is equivalent to an index into a segment of the array.

5.4.1 Alternative Implementation

A preliminary implementation of choosing among the possible versions of the method did not recompile each version separately, but rather incorporated all the code into the same method and compiled each section of the method differently. This did not work well due to interactions with other optimization passes. For example: code reordering moved some of the basic blocks so that each version was not contiguous; the register allocation for the first version affected the register allocation of the second version; and the instrumentation code was moved to incorrect locations by the instruction scheduler. In addition, this implementation strategy required that a tag be added to the control flow graph data structures to determine which instructions were created by each version. These problems were correctable, but complicated the implementation.

Chapter 6

Collaborative Server

The use of a client-server design distinguish collaborative compilation from similar systems. There are very few client-server systems in the area of compilers, and those that exist use a compilation server to compile code for a client, as in [36, 34]. This design choice leads to some complexities, such as ensuring privacy and security, but also simplifies aspects of the system and makes it more versatile. We believe that the problems with this approach can be resolved (see 6.4).

In particular, we have separated the concerns of gathering profile information and using that information. This allows the information to be persisted across runs, aggregated with similar information to create a larger knowledge base, and analyzed fully. Collaborative compilation provides a general framework for collecting profile information that allows developers of profile-directed optimizations to concentrate on implementing optimizations that can benefit from the profile information, rather than on the details of how it is collected, analyzed and published to clients.

In this chapter, we explain how data from collaborative clients is collected, aggregated, and analyzed. We also describe how improved compilation policies are generated and published to clients. We describe the difficulties in accounting for the trade-off between running time and compilation time in the virtual machine, and how this limits the analyzer and compilation policies. In addition, we address issues of privacy and security raised by the client-server design. Finally, we describe our implementation of the server and the details of how data is transferred.

6.1 Collecting and Aggregating Data

The collaborative server receives and stores data items from many collaborative clients. Each data item consists of the following information:

- Raw instrumentation data for each version and redundant copy
- Compilation-time measurements
- Machine and operating system characteristics
- The optimization that the data relates to
- A text descriptor or identifier for the compilation policy used for each version
- A list of features

This information contains everything in a MethodData object (see Figure 4-2) except the temporary values, and also identifies the optimization. The machine and operating system characteristics are included with the features.

After the data has been collected from clients, it must be analyzed to extract information useful for future compilations. The first step is to aggregate and condense the data to rectify disagreements between clients and produce labeled examples that are easier to process for an analyzer, described in Section 6.2.[1]

Running-time data is aggregated separately from compilation-time data, but they are treated in much the same way. Any differences are explained.

For each data item, if the running-time data is acceptable (see tests in Subsection 4.2.6), we compute a performance ratio. The performance ratio (or running time ratio) indicates the factor by which one compilation policy outperformed the other. To compute this ratio, first we find the mean value for both versions by dividing the total sum for the two copies of that version by the total count of that version. We then divide the mean value for version 1 by the mean value for version 2. We use

[1]An analyzer that assigns weight to the data points based on their time-stamp or the variability among clients will not use this aggregation step. However, every analyzer we have implemented has used this process to condense the data.

the ratio so that all measurements are relative (see Subsection 4.2.2). We can then aggregate this measurement with others.

As an alternative, the client can perform filtering and compute the performance ratio, instead of the server.

6.1.1 Distinguishing by Features

We use the feature vector for each data point to sort it with other data points with the same feature vector. For running-time data, we add the two compilation policies as additional features, so that each group of data points compare two compilation policies. For compilation-time data, we add the single compilation policy that produced the measurement as an additional feature.

This illustrates why it is important to include enough features. The features must describe the compilation problem well enough so that different methods can be distinguished. This is not specific to automatically-generated heuristics; a human-generated heuristic with the same information would not be able to distinguish the methods either.

At this point, outlier elimination is performed for compilation-time data, as described in Subsection 4.3.2. This removes the highest and lowest 10% of the data points within each group.

6.1.2 Computing the Mean

We use the mean value of the measurements in each group as the label for that group. For the compilation times, we will use the standard mean; with the running-time ratios, we will use the geometric mean.

In some of our experiments, we used a t-test to make sure that we had a good estimate of the mean (the t-test is described in [35]). Details for each experiment are given in Chapter 7. As we are using the geometric mean of the running-time ratios, we perform the t-test with respect to the logarithms of the ratios.[2]

[2]The test may need to be tweaked to account for median filtering or outlier elimination.

6.2 Data Analysis

The collaborative server is responsible for collecting and analyzing the client data and publishing machine-generated heuristics. We have discussed in Section 6.5 the means of collecting data and publishing heuristics. However, there are many ways to do the analysis. In general, the data analyzer must take a set of data with a single label for each feature vector and produce a heuristic as output. The feature vector will contain the extra components mentioned in Subsection 6.1.1. In addition, if the client must connect to the collaborative server to retrieve the compilation policies to test, as described in Section 5.3, the analyzer must provide responses to these requests.

Most of the components of collaborative compilation are generic and can be used for any optimization or method of analysis, including the measurement, compilation framework, data collection framework, and policy download. However, the compilation policies and analyzer must be specialized to a certain degree to the optimization being tuned. At the basic level, the analyzer must be aware of the number and type of features that the optimization uses. The compilation policy used by the optimization must agree with what the analyzer produces. Although this dependence makes the system more complex, it also makes it very extensible.

To give a concrete example of an analyzer, we have used supervised machine learning to learn a compilation policy from the performance data. This works in much the same way as described in Section 2.6. Many researchers have successfully used supervised learning strategies for compiler problems, as discussed in Subsection 9.2.1. Any of these techniques could be used with collaborative compilation.

We have implemented several procedures for data analysis, which are described in the following subsections. Our implementation of policy search is fully automatic. The profile database and supervised learning require a manual step, but this could easily be automated.

Although the analysis is performed on the server, the resulting compilation policy is used in the client. Currently there are three types of policy functions that have been implemented in the client: a vector of values, a function with numeric or binary input and output, and a hashtable lookup.

6.2.1 Profile Database

Profile information is useful for knowing exactly what happened on past runs, so that decisions can be made on the current run based on this information. In this case, we usually do not desire that the data is generalized to be applied to other methods. Instead, we want a hashtable lookup for a specific method to get information about that specific method.

To create a hashtable lookup compilation policy, our analysis consists of simply using the features of the method as the key into a hashtable, with the performance data regarding that method as the associated value. The performance data must include both the runtime ratios between the compilation decisions and the compile times for each compilation decision, so that the client can make a decision based on the expected running time of the method. The compilation policy produced by this process is the hashtable.

As an alternative, we can find the best compilation decision for each feature set, so that the hashtable maps from features to the compilation decision that the client should use. For reasons that will be explained in Section 6.3, we will ignore the compilation time data. We compare running time ratios between one compilation decision and every other compilation decision, and choose the most favorable compilation decision.[3]

Because there could potentially be many methods in our dataset, we focus on those where there is a clear benefit to one compilation decision. Our implementation requires that there is at least a 3% difference in performance between two of the compilation decisions. In addition, our implementation requires that a fixed number of compilation decisions are included in the performance data. It also requires that

[3]We performed sanity checks to ensure that the measured ratios were transitive. I.e., if the ratio between A and B is $A/B = r_1$ and the ratio between A and C is $A/C = r_2$, then the ratio between B and C is $B/C = r_2/r_1$.

the sole feature is a hash of the method name and signature. This implementation was used as a proof of concept, described in Section 7.1.

6.2.2 Policy Search

Using policy search is useful when there are too many possible compilation decisions for an exhaustive comparison to work. For example, if a parameter can be any integer between 100 and 1000, each method would need to be tested 899 times to get even a single label, which may be incorrect due to noisy measurements. Policy search is also useful if several decisions must be made for each method, and the performance is dependent on the interactions between the decisions. For example, even though there are only two possible compilation decisions for inlining (inline or don't inline), it is difficult to measure the impact of a single decision, since the performance is dependent on whether the calls before and after are inlined. Any conclusions drawn from this performance data will be adapted to the scheme used to decide the other inline sites, and may not work well when it is used to make all the inlining decisions in the method.

We have implemented two types of policy functions that can be used for policy search: functions with boolean and real-valued inputs and output, and vectors of categorical or numeric values. The server implementation of policy search is very similar for these two policies, but they have somewhat different usage on a client.

The function policy is a reimplementation in Java of the priority functions used in [43], described in Subsection 2.5. However, our implementation does not support crossover, and therefore is not a form of genetic programming, but rather a form of hill-climbing (also known as gradient descent). The function is an expression tree with numeric and boolean operators (internal nodes) on constants or features (leaf nodes). To create a new expression of a given type, an operator is probabilistically chosen and subexpressions are recursively created until reaching the depth of the tree, where terminals are probabilistically chosen. To mutate itself, the expression replaces a random node with a leaf or tree of the same type. The depth of the tree begins at a specified value but may change with mutation. The expression can be serialized into text as a fully-parenthesized expression in Polish notation, and the parser can recreate the expression on the client. The expression can be evaluated by providing an array of the features.

The value vector policy is simply a vector of categorical or numeric values which can be used to set options or parameters in the compiler. In the client, it can be used either to look up values of variables in a hashtable, or to modify an object containing the compilation options. The possible range of values for each component are specified in the server. To create a new vector, values are selected at random for each component. To mutate itself given a mutation frequency, each categorical value is mutated with probability equal to the frequency, and each numeric value is chosen using a Gaussian distribution with mean equal to the old value and standard deviation proportional to the frequency and the old value. The vector can be serialized into text as a comma-separated list, and the parser can recreate a hashtable on the client.

Either of these policies can be used with the hill-climbing analyzer. The algorithm consists of a number of rounds, where two policies are compared in each round. In the first round, two initial policies are created. In our experiments, we seeded one of the initial policies with the optimization's default policy, but the default policy usually disappeared within a few rounds. The other initial policy was randomly generated. Clients will then connect to the Broadcast Server to retrieve the current two policies and test them. After collecting performance data, we select the average running time ratio that compares the current two policies. We decide if there is enough data by some method, detailed in the next paragraph. If we lack enough data, we continue to collect data. Otherwise, the best policy is kept and another is created by mutating the best policy. These become the new current policies and the process continues. At any point, the current best policy is the last winner, which can be published for clients to use.

We implemented two ways to check if there is enough data. The simplest starts a new round after a fixed number of data points (ratios). A more complicated method computes confidence intervals on the mean of the logarithms of the ratios. If the confidence interval does not include zero, we can conclude that there is a statistically significant difference between the two policies. If the entire confidence interval lies within a small interval near zero (in our experiments, $\pm 1\%$), we can conclude that there is not a significant difference greater than the interval. In the latter case, we arbitrarily chose the older policy.

In Section 6.3, we discuss the problems with integrating compilation time information with the technique presented here and alternative techniques for including compile time information in policy search.

6.2.3 Supervised Machine Learning

In many ways, using supervised machine learning to analyze the performance data is similar to using the profile database. However, it can also generalize to methods for which we do not have data and mitigate the effects of errors in the data. In summary, supervised learning produces a function of the features that closely fits the measurement for those features. In addition, many machine learning techniques attempt to produce a function that will give good predictions of the measurement for features it has not seen (the test set), even though this may decrease its accuracy on the known data (the training set). A detailed explanation of supervised learning, including the techniques discussed here, is given in Section 2.6.

Given a set of labeled features, nearly any supervised learning technique can be applied to arrive at a compilation policy for the problem. Other researchers have used a wide variety of techniques for compiler problems, discussed in Section 9.2. We will restrict our discussion here to what we have implemented and simple extensions that are possible.

We used the Weka toolkit for data mining [46] to manually analyze the data. Weka can perform many types of supervised and unsupervised machine learning within the same framework. After learning a compilation policy that performed well on the dataset, we implemented this function in the compiler.

We performed most of the analysis and feedback to the client manually; however, since Weka is implemented in Java and has a good programmatic interface, it would be fairly simple to perform the machine learning automatically. Any technique that produces a function that can be represented as an expression tree from the previous section could be used on the server as an analyzer. This includes any technique that produces a linear function and decision trees. Other techniques would require only that an evaluator be implemented in the client.

6.3 Balancing Running Time and Compilation Time

As explained in Section 2.3, the virtual machine must balance compile time and running time to achieve the best performance. Unfortunately, we can not combine compilation time and running time into a single total execution time measurement without making some assumptions. If these assumptions are not valid, then the running time measurements can not be combined with the compilation time measurements on the server.

Before continuing, we should clarify how we compute total running time and total execution time for a single data point. To compute total running times, we do not use the running time ratios, but go back to the raw data collected from clients. There are four sample counts per method, for two redundant copies for each of two versions. If the sampling rate is α, the counts for each copy are n_{ij}, and the average time for each version is $\bar{x}_i$, we can get an approximation of the total running time for each version, r_i:

$$n_{samp} = n_{11} + n_{12} + n_{21} + n_{22}$$

$$n_{run} = \frac{n_{samp}}{\alpha}$$

$$r_i = n_{run} \cdot v_i.$$

That is, we add to get a total number of samples, divide by the sample rate for an approximation of the number of times the method ran, and multiply by the average time per run to get the approximate total running time for each version. We then simply add the average compile time for each version, c_i to the total running time to get the total execution time, t_i:

$$t_i = r_i + c_i$$

If we were to use a single measure of performance that includes both running time and compilation time (such as total execution time), then for each feature vector, our compilation policy would give a single compilation decision to apply to the method. This assumes that a single decision will work best, whether the method will execute only for an additional two seconds or as much as two hours. Thus, the decision of which compilation decision is best can not be based on the expected running time improvement (over the baseline) of the method being compiled, because we would require an online prediction of the baseline future running time, rather than offline data on previous running times.

We have rejected several potential solutions that allow a single solution to be used in the virtual machine, because they will work only in specific cases.

- Suppose that we can assume that the past measured running times for each method are good predictions for the future running time. We can then use the total execution time to compare compilation policies.

  In our experience using Jikes RVM, this assumption often does not hold true. The adaptive system uses online profile information to decide when to compile a method. This profile information differs in each run, so the running time of a method after recompilation is unpredictable. In effect, this tries to predict the future running time of the method based on features specific to a particular optimization, which is unlikely to succeed.

- Suppose that we assume that the total execution time can be estimated as a function of the running time ratios and compilation time ratios (a ratio between the average compilation times, similar to the running time ratios). For example, we could equally weight compilation time and running time, or assume compilation time is 10% of total execution time. This is similar to the last bullet point, except that we are no longer attempting to predict future running time. Instead, we assume a certain relationship between the compilation time and running time.

  In fact, the running time and compilation time are not related. Any predetermined weighting is arbitrary, and if the relationship does not hold at run-time, the solution will be non-optimal. However, Cavazos *et al.* have had good results for a "balance" between running time and compile time [10].

- Suppose that we assume that the choice of compilation decisions does not affect the compilation time. In this case, we can ignore the compilation times altogether and simply choose the decision with the best running time improvement.

  This assumption may work well for certain optimizations. However, we attacked optimizations where this is definitely not the case. With inlining, choosing to inline a call site roughly increases the total compilation time by the compilation time of the callee. In addition, this allows the callee's call sites to be inlined. When tuning the adaptive system, the choice of optimization level has a huge impact on the compilation time, as shown in Chapter 8.

- Suppose that we assume that compilation time does not matter. For long-running programs, this is usually true. For the best performance on long-running programs, all methods should be compiled at the highest optimization level after gathering enough online profile information to make good compilation decisions.

  We take this approach in one experiment, described in Subsection 7.1.3, and achieve good results. However, this tends not to be the common case, nor is it the interesting case.

Although these assumptions may hold in specific domains, they are not helpful for the general case. We used two techniques in our experiments to overcome this problem.

- In Subsection 6.2.3, we predict compilation time based on the features of the method and the compilation decision under consideration. This approach may have limited applicability.

- In Subsection 6.2.1, we include compilation time data in the profile information that is sent to the client. The client can use the online future running time estimate to compute the best trade-off between compilation and running time.

As an approach that could have wider applicability, we could provide two heuristics for each optimization in the virtual machine that we tune. One heuristic would give the performance ratio for a particular compilation decision and the other heuristic would give the increase in compilation time by using that compilation decision. We could then use the best online estimate of future running time to make a choice about which compilation decision to use. This seems to be a general approach; however, it requires two heuristics for each optimization, which makes it quite a bit harder to use. We leave this as a topic of future work.

6.4 Privacy and Security

As with any application that transfers data across the network, privacy and security are important issues that must be addressed for collaborative compilation. Data about the applications that users run is sent to a potentially untrusted server, so users may be wary of information leak. In addition, since the compiler controls the code that is run by the computer, it must be protected from intruders.

Our prototype implementation of collaborative compilation is not focused on privacy and security. Nevertheless, our implementation provides some security by virtue of its design. A collaborative compilation policy controls only the performance of an optimization, not its correctness. The outside compilation policy is not allowed to jeopardize correctness. In addition, the client parses and interprets the compilation policies, so no executable code is transferred.

Information from the client that is transmitted to the server does not explicitly identify the program or data, only the features of the code that is compiled. However, this does not necessarily mean that the server can not deduce this information; the program may leave a "fingerprint" in the data that is sent, such as a feature vector that is unique to a certain program or a distribution of features over several runs that match the program. Thus, feature vectors of individual methods that the client runs should not be transmitted to an untrusted collaborative server.

This complicates implementation of a supervised learning strategy; this is discussed further in Subsection 10.2.2. Policy search is an attractive alternative, as very little information needs to be sent to the collaborative server. The client need only report which of two compilation policies it prefers. It is conceivable that the compilation policy could be engineered to provide very bad performance for methods with a certain feature vector, but this information leak is very small.

In any case, a user could run a collaborative server locally, which alleviates all privacy concerns. In this case, the system reduces to the traditional profile-directed approach, since no profiling information is shared. Directions for future work regarding the privacy and security of the system are discussed in Subsection 10.2.2.

6.5 Text-Based Server

For simplicity, the collaborative server is largely text-based. It is comprised of two similar servers that allow clients to send data to the server and receive compilation policies from the server. The protocol is exceedingly simple: the client connects to the server using TCP sockets, the text is sent, and an acknowledgment is returned. In the case of the Data Server, the client sends the server information; in the Broadcast Server, the roles are reversed.

6.5.1 Data Message Format

The client data is formatted as comma separated values. It contains several fields of raw instrumentation data for each version, compilation-time measurements, possibly machine and operating system characteristics, the optimization that the data relates to, a text descriptor or identifier for the compilation policy used for each version, and a list of features.

When a client submits new data, the data server will save the data to disk and process the performance data. It runs the filters mentioned in Subsection 4.2.6 to ensure that the new data is acceptable. The data i then aggregated into the knowledge base, as described in Section 6.1.

6.5.2 Policy Message Format

The policy message consists of two compilation policies and their unique identifiers., described below.[4] Each compilation policy is serialized as text by the server and parsed by the client. This helps to ensure security, and makes the transmission human-readable.

With a variant of the Broadcast Server, a client can request a heuristic tailored to that client's machine, to a specific program, or to a certain usage pattern. This option could also be used to specify which optimization should be tested if several optimizations are being tuned at the same time.

This concludes the description of our implementation of collaborative compilation. In the next two chapters, we show empirically that the techniques presented in Chapter 4 work well, and that we can achieve good performance improvement using the components described in Chapter 5 and in this chapter.

[4]A compact compilation policy descriptor can act as its identifier.

Chapter 7

Experimental Methodology

We present the methodology used to arrive at our experimental results, given in Chapter 8. Our first experiments test the quality of our system of measurement. We check that our techniques for noise reduction work well and that the overhead is low. We present two proof-of-concept experiments using a profile database analyzer. We then explain our methods to use supervised machine learning as the analyzer to predict compilation time.

7.1 Quality of Measurements

This section describes the experiments that we performed to validate our approach to measurement, presented in Chapter 4.

7.1.1 Effectiveness of Noise Reduction

To test the effectiveness of the techniques for filtering noisy measurements presented in Section 4.2, we use the techniques to measure a known quantity. When both versions of a method are compiled using the same compilation policy, as described in Chapter 5, then the resulting methods should have average running times that are the same. Since the two versions have the same mean running time, the running time ratios for each method should be near one.

To perform this experiment, we ran the benchmarks from the DaCapo benchmark suite [18] using the default settings for the Jikes RVM compiler for both versions. We collected thousands of running time ratios while running the system with and without median filtering and filtering with redundant copies. In Section 8.1, we show these techniques result in a far greater percentage of the running time ratios being near one.

7.1.2 Overhead of Measurement

An important goal of collaborative compilation is being unobtrusive to the user. This implies that collaborative compilation can not cause a large overhead. We computed the average overhead of the system over twelve complete runs of the specjvm98 benchmark set [41]. We measured the overhead numbers using the user time reported by the `time` command. These numbers include the cost of compiling four additional methods for each collaboratively instrumented method and connecting to the knowledge base over the network.

In the next section we discuss an experiment where our prototype compiler automatically tests the efficacy of two key parameters for tuning the method inliner in Jikes RVM. The effectiveness of these settings is highly dependent on which method is being compiled. As a result, collaborative compilation will sometimes try a setting that reduces the performance of the instrumented method. Our measurements of the overhead, presented in Section8.2, include the overhead of occasionally making bad decisions.

For all the experiments we perform, we vary the percentage of methods that we consider for instrumentation, but we cap the number of instrumented methods at one. In other words, we never instrument more than one method per run. Our results show this alternative as well.

7.1.3 Profile Database

This section serves as a proof-of-concept validation of our infrastructure. The first experiment investigates improving the steady state performance of the DaCapo benchmark suite [18]. The second experiment attempts to minimize the total execution time of a benchmark by tuning the virtual machine's adaptive system. These experiments provide important feedback on the accuracy of our infrastructure's measurements.

Improving Steady State Performance

As described Section 6.3, some users are more concerned with steady state performance than they are about compilation overhead. This section describes a simple experiment with the method inliner in Jikes RVM. Method inlining (described in Subsection 2.1.1) can have a large effect on the resulting performance of a Java application. Usually the effect is positive; however, our system found that inlining the method calls of the primary method (`TrapezoidIntegrate`) of `series` in the Java Grande benchmark suite [21] leads to a 27% slowdown.

There are two key inlining parameters that adjust the aggressiveness of Jikes RVM. One parameter sets the maximum inline depth, and the other restricts the size of methods the inliner will consider for inlining. In this experiment we used our system to compare the relative benefits of three options: using the default settings, doubling the value of both parameters, and halving the value of both parameters.

Using the technique in Subsection 6.2.1, we created a profile database for each program in the DaCapo benchmark suite (version 050224) [18].[1] This experiment was performed on a single 1.8 GHz Pentium M machine with 512 MB of memory running Ubuntu Linux. Because we are concerned with steady state performance, we compare against compiling all methods at the highest level of optimization. We also use the so-called `replay` system in the virtual machine, which is essentially a profile-directed system. We verified that aggressive compilation and the use of profile-directed feedback in the comparison system is faster (in the steady state) than

[1] At the time that this experiment was performed, our version of Jikes RVM could not run `xalan`, `batik`, or `chart` because of preexisting bugs.

allowing the virtual machine to incrementally re-optimize methods. To collect the steady state runtime, we iteratively run each benchmark three times and take the fastest of the last two runs. It is important to note that in `replay` mode, the VM compiles all methods upfront, and thus compilation time does not factor into these results (*i.e.*, no compilation is performed in the second and third iterations of the benchmarks).

The results of this experiment are presented in Subsection 8.3.1.

Tuning the Adaptive System

This experiment attempts to minimize the total overall runtime by tuning the adaptive system in Jikes RVM. Recall from Section 2.3 that the adaptive system evaluates the costs and benefits of recompiling hot methods at higher levels of optimization. It tries to find an optimization level for which the benefits, *i.e.*, reduction in running time, outweigh the costs, *i.e.*, compilation time.

In this experiment, we see how well the adaptive system performs if, for a given method, it had accurate information about the compilation time and running time improvement of the four optimization levels (`O-1`, `O0`, `O1`, and `O2`). We use our collaborative system to automatically gather this data. We gathered our data on a cluster of workstations that is shared by several students. The load on these machine varied irregularly during the data collection phase, thus giving realistic, noisy usage conditions.

This experiment uses a *t*-test to determine that there are enough samples to accurately specify the running time ratios, as described in Subsection 6.1.2. The system determines the confidence intervals to ensure that we have enough samples to specify the runtime ratios within $\pm3\%$ with 90% confidence.

We created a profile database, as described in Subsection 6.2.1, to store the costs and benefits for each method. For each method, the database stores the empirically-found compilation times for each optimization level and the improvement in running time from one optimization level to the next (the running time ratio). When the virtual machine runs it consults the database, which is currently implemented as a file on an NFS system, to read in its application-specific hashtable. The collaborative compiler can then make optimization decisions based on more accurate running time ratios and compilation times.

As described in Section 2.3, the adaptive system estimates that a method's future running time will be equal to its current running time, each optimization level has a fixed compilation rate, and the running time ratios between optimization levels are independent of the method being compiled. Our updated version of the recompilation heuristic uses the method-specific cost and benefit numbers to make more informed decisions. Our heuristic still assumes that the remaining runtime will be equal to the current runtime. This experiment is orthogonal to Arnold *et al.* who maintain a repository of past profile information, which allows for better predictions of future runtime [5] (see Chapter 9).

We gathered data for the DaCapo benchmark suite [18] using the "default" input data set only. We then measured the performance of the same benchmarks using the "default" data set and the "large" data set, drawing profile information from the database. The results presented in Subsection 8.3.2 include the time it takes to access the local database.

7.2 Predicting Compile Times

This section describes an experiment to improve the adaptive system of Jikes RVM by providing accurate predictions of the compilation times of methods. We used supervised machine learning techniques to learn a function from features of the method to the compilation time of the method. In this experiment, we used Ridge Regression, a supervised learning technique described in Section 2.6.

7.2.1 Features

In order to predict method compile time using machine learning, we will need to collect features of the method that may be related to the compile time. Since the adaptive system must choose an optimization level before the method's control flow

Feature	Description
hasSynch	Whether the method acquires locks
hasThrow	Whether the method throws exceptions
hasSwitch	Whether the method contains a switch statement
sizeEstimate	An estimate of the size of the method when inlined
numAllocation	Number of object allocations in the method
numInvoke	Number of calls in the method
numFieldReads	Number of reads from object fields in the method
numFieldWrites	Number of writes to object fields in the method
numArrayReads	Number of array reads in the method
numArrayWrites	Number of array writes in the method
numFwdBranches	Number of branches to a higher bytecode index
numBkwdBranches	Number of branches to a lower bytecode index
localVarWords	Number of words used for local variables and parameters
operandStackWords	Maximum number of words needed for intermediate values on the stack
bytecodeLength	Number of bytes in the bytecodes for this method

Table 7.1: Features for Predicting Compilation Time. We used these values as features for predicting compilation time. Because we must collect the features before any intermediate representation is created, they may not be ideal, but they estimate factors that likely contribute to compilation time, as described in Subsection 7.2.1

graph has been created, we are limited in the features that are available. However, we can gain some information about the method by scanning the bytecodes that are to be compiled.[2] Table 7.1 lists the features that are available to the machine learning algorithm. These include features that summarize the complexity of the control flow, the number of memory operations, and estimates of the eventual number of instructions in the method. Some features are only approximations for values we can not find until later in the compilation process. For example, we include the number of backward branches as an approximation for the number of loops and loop nesting depth, which are not available until the control flow graph is created. The number of calls approximates the additional instructions that will be added to this method as a result of inlining these calls, which will not be known until after the inliner has run.

[2]Jikes RVM calculates an estimate of the size of every method for the inlining optimization by scanning the bytecodes as they are loaded. Therefore, the features can be collected without incurring any additional cost.

7.2.2 Data Collection

In order to use machine learning to predict the compile time based on these features, we first need to collect data on the compile time. We run the collaborative compiler, specifying an optimization level as our compilation policy. We chose to compare level O0 against O1 and O2, so the two compilation policies are either O0 and O1, or O0 and O2. The collaborative system compiles one version of the method at each of these two optimization levels. The time taken for the optimizing compiler to compile each version of the method is recorded along with a vector of the features in Table 7.1 and the optimization level at which the method was compiled. The clients send this data to the collaborative server.

To simulate collaborative users, we ran benchmarks on a cluster of ten machines using the collaborative virtual machine. As mentioned, we used supervised machine learning as our analyzer, which is described in Subsection 6.2.3. Although this step was performed manually in this experiment, the process could be automated to provide collaborative users with the most current compile time predictor with little human intervention.

We used the process described in Section 6.1 to aggregate the compilation time data, and assign a single compilation time label for each example method in the training set. We aggregated based on the features and computed an estimate of the true compile time. Using the technique described in Subsection 4.3.2, we removed the outliers by dropping the highest and lowest 10% of the compilation time measurements. This removed all outliers and improved our estimate of the true compile time. We computed the mean and standard deviation of the remaining samples and performed a t-test to determine if the 90% confidence interval was within 5% of the mean. If the t-test did not fail, we created a machine learning data point labeling the features of the method with the mean compile time.

Recall from Section 4.3 that in order to collect data from a variety of machines, we measure compilation time relative to the baseline compiler. Since we collected data on identical machines, this variability is not a concern, and we did not normalize by the baseline compilation rate.

7.2.3 Ridge Regression

Subsection 2.6 describes Ridge Regression, a fairly simple supervised learning technique that finds the linear equation that best fits the data, while also performing feature selection. Ridge regression is simple, efficient, and worked well on all of our data sets, after applying the space transformation described in Subsection 7.2.4. We experimented with using a support vector machine regression algorithm and found that it performed marginally better; however, the increased complexity and time for prediction did not warrant its use. We also experimented with a decision tree algorithm, but it performed poorly.

7.2.4 Space Transformation

The ridge regression was able to fit the compile time data very well for `O0`. The compile time is largely linear in the number of bytecodes, which is the model that Jikes RVM assumes. However, for `O1` and `O2`, ridge regression does not do as well, for several reasons. First, the compile times for `O1` and `O2` has far more variance than those for `O0`, making prediction difficult for any regressor. Second, some of the compile times are very large—orders of magnitude greater than the mean—while most are fairly small. The regression will try to fit the large compile times to reduce the large error by sacrificing accuracy for the small values. Even if the compile times are normalized to be in the range $[0, 1]$, the large values will have much more influence than the smaller values. Third, the distributions of the compile time and nearly all the features are skewed toward small values, simply because there are far more small methods than large ones. The regression has trouble fitting the densely packed small values as well as the more sparse large values.

We solved the problems of skewed distribution by transforming the features (except boolean features) and the compile time using logarithms. Taking the logarithm expands the space of the small values and contracts the space of the large values so that the data points are more evenly distributed. It also drastically reduces the large values and decreases the squared error so that the regression is not pulled toward those values. In this transformed space, the ridge regression can make a surprisingly good fit.

The log transform regression is analogous to a regression in the original space where the error is measured in units of the actual compile time. That is, if y_a is the actual compile time and y_p is the predicted compile time, the relative error is

$$\frac{y_p - y_a}{y_a} = \frac{y_p}{y_a} - 1.$$

Minimizing actual error in the log space will minimize the relative error in the original space. Minimizing relative error may be more sensible for compile time; for example, predicting that a method will take 11 milliseconds to compile when actually it will take 1 millisecond causes worse performance than predicting 60 milliseconds instead of 50.

Chapter 8

Results

We generally had good results from the experiments described in Chapter 7. This indicates that our prototype collaborative system can be used further to provide additional performance benefits.

In this chapter, we begin by validating our measurement techniques, then move on to show that we can improve performance using these measurements. The experiments generally increase in complexity. We start by using the collaborative system to create a profile database that improves steady-state running time. Another experiment obtains good results for total execution time using a similar technique. Finally, we conclude with an experiment that uses supervised machine learning as the analyzer in the collaborative system, also improving total execution time.

8.1 Higher Precision Measurements

As described in Subsection 7.1.1, we performed an experiment to test our measurement accuracy. We check the accuracy with and without median filtering and redundant copies, two noise filtering techniques described in Section 4.2. We compared thousands of identical versions to check that the measured values are the same, *i.e.*, that the ratio between the two values is near one.

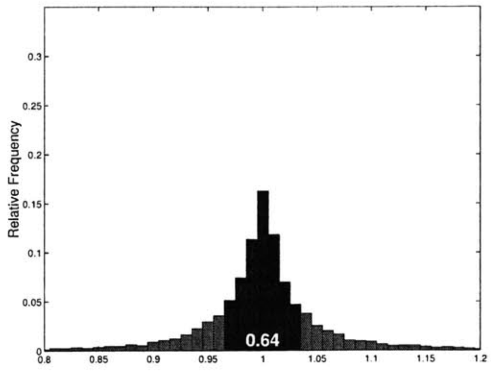
As can be seen in Figure 8-1, these methods do eliminate a great deal of noise. These histograms empirically demonstrate that our system, using median filtering and redundant copies, can effectively reduce measurement error. Notice that most of the ratios are tightly clustered around 1.0, which is the desired value for this experiment. In addition, each histogram is symmetric around the correct mean, indicating that our measurements are unbiased.

The figure is especially compelling because the data comes from machines that are shared by several research groups, and the load on each machine varied unpredictably during the experiment. The data in Figure 8-1(a) shows that no filtering of the data yields poor results. Only 64% of the data is clustered within 3% of 1. This could be due to any of the sources of noise discussed in Section 4.1. Part (b) shows that median filtering can provide modest improvements in accuracy. The use of redundant copies is by far the most effective technique for reducing noise, as part (c) indicates. Finally, by combining median filtering and redundant copies, our system ensures that 89% of the running time ratios are within ±3% of 1. Almost all of the data (96%) is clustered within a 5% noise margin. This indicates that although their is a great deal of noise from many sources, we are able to effectively filter the noise to get good measurements.

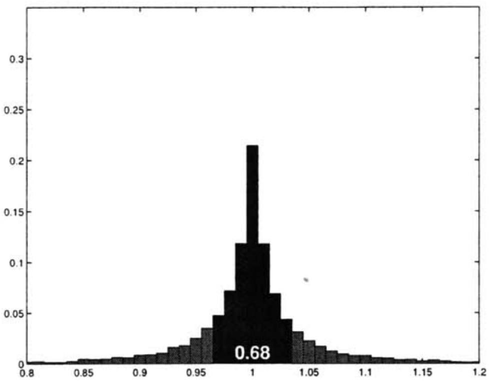
8.2 Low Overhead

This section discusses the overhead associated with collaborative compilation. Table 8.1 lists the average overhead of the system over twelve complete runs of the specjvm98 benchmark suite [41]. These overhead measurements include the extra time required if the collaborative compiler makes a bad decision, as well as the extra time required to transmit data to the collaborative server. As we can see, the overheads are quite low. The Rate column gives the percentage of methods that the system will collaboratively compile. In other words, the rate can be adjusted to control the probability that any single method will be instrumented. The last row shows the overhead when only a single method is compiled using our system.[1] We ran all

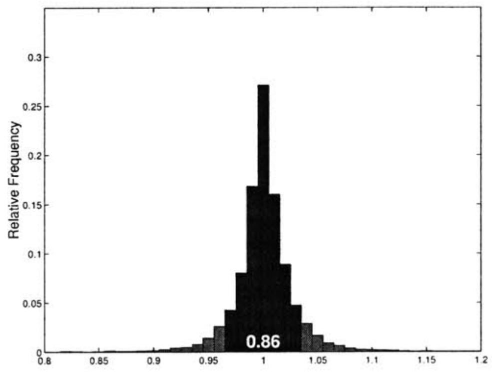
[1]The probability of choosing any single method can be adjusted here as well. However, most settings will yield the same behavior: one method is compiled. The overhead in the last row was measured with a rate equal to 0.10.



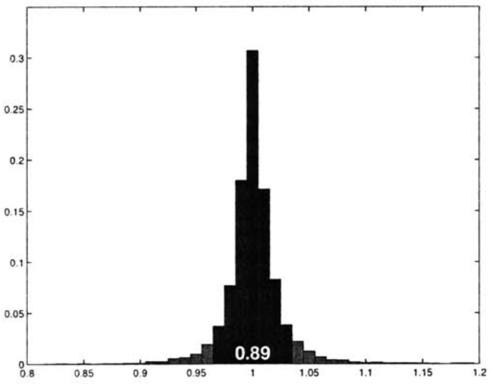
(a) No filtering.



(b) Median filtering.



(c) Redundant copies.



(d) Redundant copies & median filtering.

Figure 8-1: Higher Precision Measurements. This figure shows histograms of running time ratios of thousands of identical methods. A noiseless measurement would produce exactly 1.0 for this experiment. Each histogram shows the distribution of the measured running time ratio using different filtering techniques. The number within the histogram shows the percentage of measurements within our 3% noise margin. (a) no filtering, 64% within noise margin. (b) median filtering only, 68%. (c) redundant copies only, 86%. (d) both median filtering and redundant copies, 89%.

Rate	Total
0.01	1%
0.05	8%
0.10	13%
0.20	24%
One method only	2%

Table 8.1: Low Overhead of Collaborative Compilation. This table gives the run-time overhead of collaborative compilation averaged over 12 complete runs of the specjvm98 benchmark suite [41]. The rate column shows the rate at which methods under compilation are selected to be collaboratively compiled. The total column contains the total overhead of the collaborative system, including the cost of compiling redundant copies, and of sending any gathered data to a collaborative server. The last row shows the overhead when at most one method is instrumented per run.

our experiments using the final option, instrumenting a single method per run.

8.3 Performance Gain using Profile Database

Collaborative compilationworks successfully when using a profile database of running time and compilation time measurements to improve the predictions of the adaptive system. We obtained positive results for the experiments described in Subsection 7.1.3. The following subsections give detailed results.

8.3.1 Improving Steady State Performance

As described in Subsection 7.1.3, we used the collaborative system to create a profile database for inlining decisions as a proof-of-concept experiment. Figure 8-2 shows the gains attained by our system on the DaCapo benchmark suite when using a profile database of running time ratios between three settings of the inlining parameters. The results are averaged over ten runs of each application and input data set. The "default" input data set was exclusively used to populate the collaborative database. The black bar indicates the performance of each application on the default data set. We see that the collaborative database leads to substantial improvements for four of these benchmarks, and furthermore, it does not slow down a single application. Interestingly, when we apply the alternate dataset, the performance is actually better

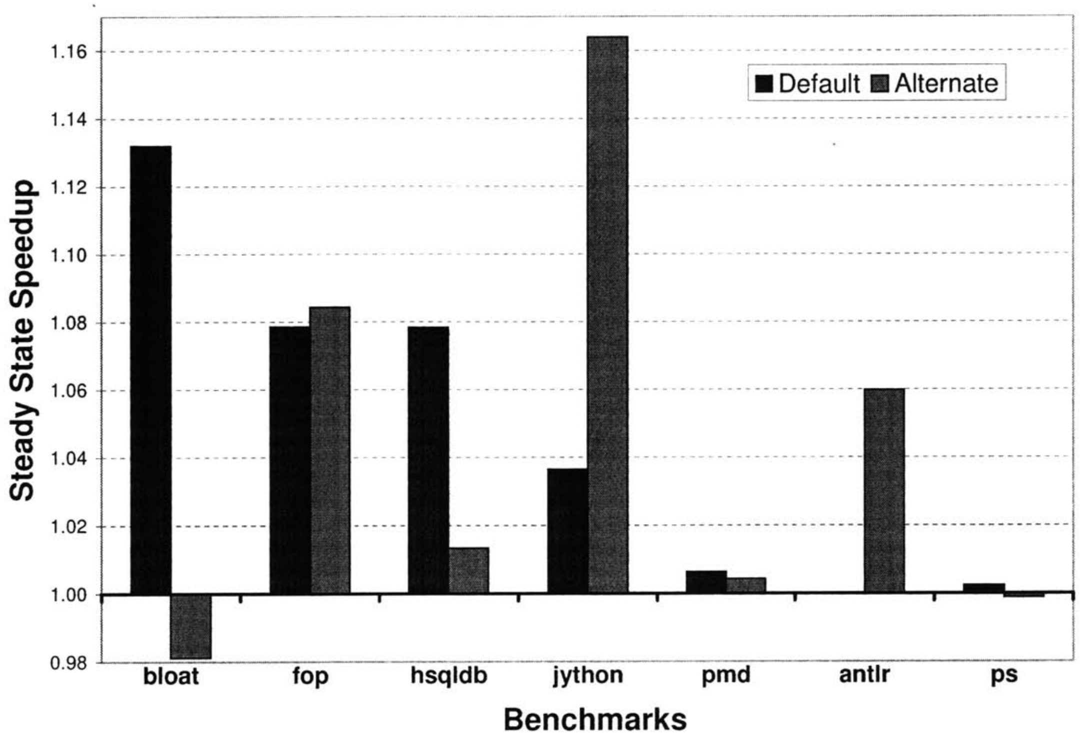


Figure 8-2: Improving Steady State Performance. This figure shows the performance gain when using the profile database to improve steady state performance (*i.e.*, reduce running time). The database was created by running the collaborative compiler with each benchmark using the "default" input. As we can see, the benchmarks are improved when using the profile database on the same input. When using the alternate dataset, we realize the same performance gains.

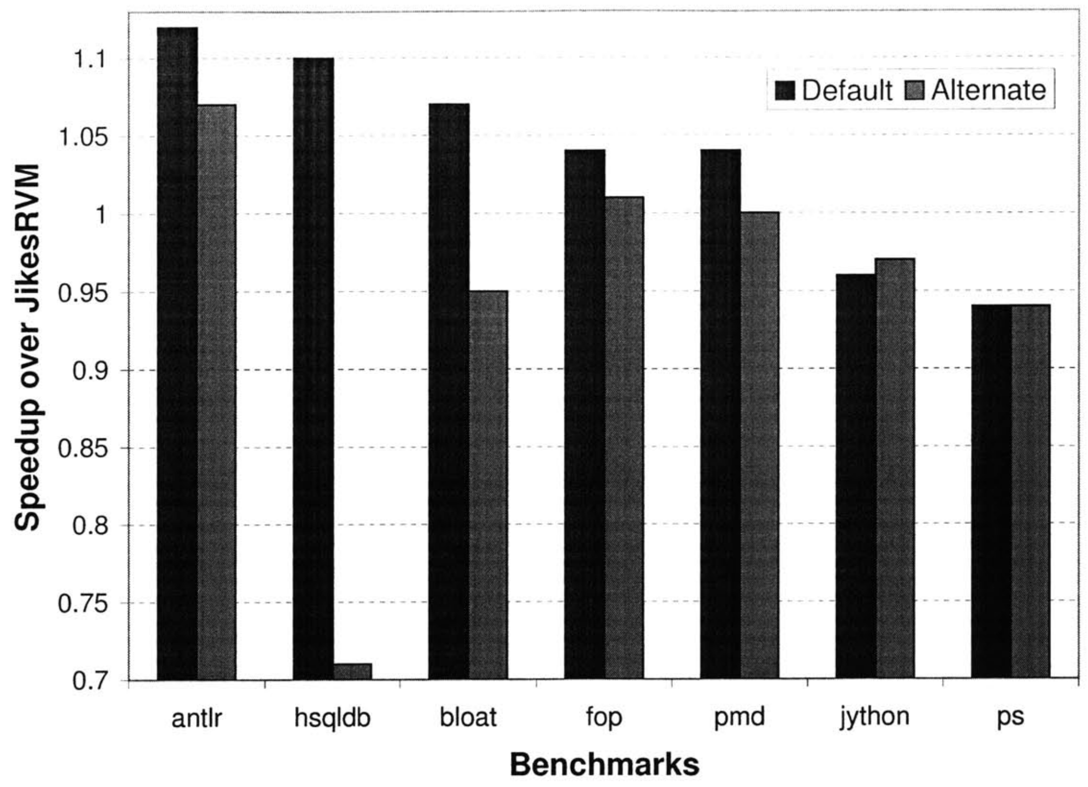


Figure 8-3: Tuning the Adaptive System. This figure shows the performance gains and losses when using a profile database to tune the adaptive system. The "default" dataset was used as input to create the profile database. This figure shows the results of using this database when running the default dataset and an alternate dataset.

for three of the benchmarks.

8.3.2 Tuning the Adaptive System

By using a profile database containing the running time ratios and compilation times of methods, as described in Subsection 7.1.3, we were able to improve the performance of the DaCapo benchmark suite [18]. Because this experiment considers total execution time, rather than only running time, it is a much more difficult task than the experiment above in Subsection 8.3.1.

We trained the collaborative compiler to minimize the total execution time of the seven benchmarks with the "default" input data set only. Figure 8-3 shows the results attained for this experiment. The dark bars in the figure show the factor of improvement on the default input data set. Because all of the data that the system collected was extracted using this dataset, it is not surprising that we improve performance for the majority (five out of seven) of these benchmarks.

Although our system hinders the performance of `jython` and `ps`, we feel that these are good results, considering the difficulty of balancing running time and compilation time. The slowdowns may be due to poor compilation time estimates, as we had not yet implemented the improved compilation time estimates, discussed below in Section 8.4.

The lighter bars in Figure 8-3 show the results of our system when a different and significantly larger input data set is used for each benchmark. The fact that performance is much worse with the alternate data set is an argument in favor of collaborative compilation. Previous attempts to learn compiler heuristics have performed at-the-factory training, which means the compiler developer chooses which programs and on which input data sets the compiler will be trained. As this figure shows, performance can vary wildly from one input set to another. In complex, object-oriented applications, bad profiling information can be extremely detrimental. For instance, inlining a virtual method might be advantageous for methods of a given class (that are exercised in the training set), but detrimental for other classes. A collaborative compiler has the potential to adapt to unfamiliar program usages.

8.4 Compile Time Prediction Accuracy

In Section 7.2, we described an experiment to accurately predict compilation times based on features of the method to improve the performance of the adaptive system in Jikes RVM. Here we describe the results of that experiment.

We ran the collaborative system on a cluster of ten 1 Ghz Pentium III machines running Red Hat Linux with kernel 2.4.21–37.ELsmp. We repeatedly ran benchmarks from the DaCapo benchmark suite [18].[2] We collected approximately 26 thousand

[2]This experiment used Jikes RVM version 2.4.2, which is not able to run the benchmarks `batik` or `chart` due to an incomplete implementation of the Java libraries, and fails intermittently with the `hsqldb` benchmark for unknown reasons. In addition, we were not able to extract the compilation

Feature	O0	O1	O2
sizeEstimate	0.9627	0.7539	0.6985
bytecodeLength	0.9316	0.6142	0.6555
numInvoke	0.9295	0.7839	0.7398
numAllocation	0.8354	0.8278	0.6569
numFwdBranches	0.8197	0.4650	0.5247
localVarWords	0.5169	0.1222	0.2233
numArrayReads	0.4967	0.1222	0.1741
numFieldReads	0.4809	0.2320	0.1757
numBkwdBranches	0.4688	0.1737	0.2301
hasSwitch	0.4002	0.2447	0.3742
operandStackWords	0.3311	0.0531	0.1339
numArrayWrites	0.2039	0.0440	0.0635
numFieldWrites	0.1532	0.0355	0.0620
hasThrow	0.1274	0.0683	0.0912
hasSynch	0.0538	−0.0190	−0.0259

Table 8.2: Correlations of Features with Compilation Time. This table lists the correlations between each feature and compilation time. This measure can be used to rank how well the feature could predict compilation time on its own. For each feature in the first column, the remaining columns show the correlation with the actual compilation time for each of the three optimization levels. The features are ranked in order of the correlation for O0. The most correlated features for the other two optimization levels differ from O0.

samples from clients. A large number of these were removed by filtering. Within the remaining data set, there were 291 distinct feature vectors (methods) with compilation time labels for optimization level O0, 237 for O1, and 194 for O2. These include methods from the following benchmarks: `antlr`, `bloat`, `fop`, `jython`, `pmd`, `ps`, `xalan`.

As a pedagogical exercise we begin by identifying the features that are most strongly correlated with the true compile times. Table 8.2 shows the results of this experiment. The second row in the table shows the sole feature that the Jikes RVM adaptive system uses to predict compile times, bytecodeLength. We can see that this feature is fairly effective for predicting compile times of the O0 compiler. However, for higher levels of optimization, it is not a good predictor of compile time.

Figure 8-4 plots the log of the bytecode length versus the log of the compile time. This chart graphically shows that while bytecode length is a good predictor

time predictions for the ps benchmark due to an unknown bug.

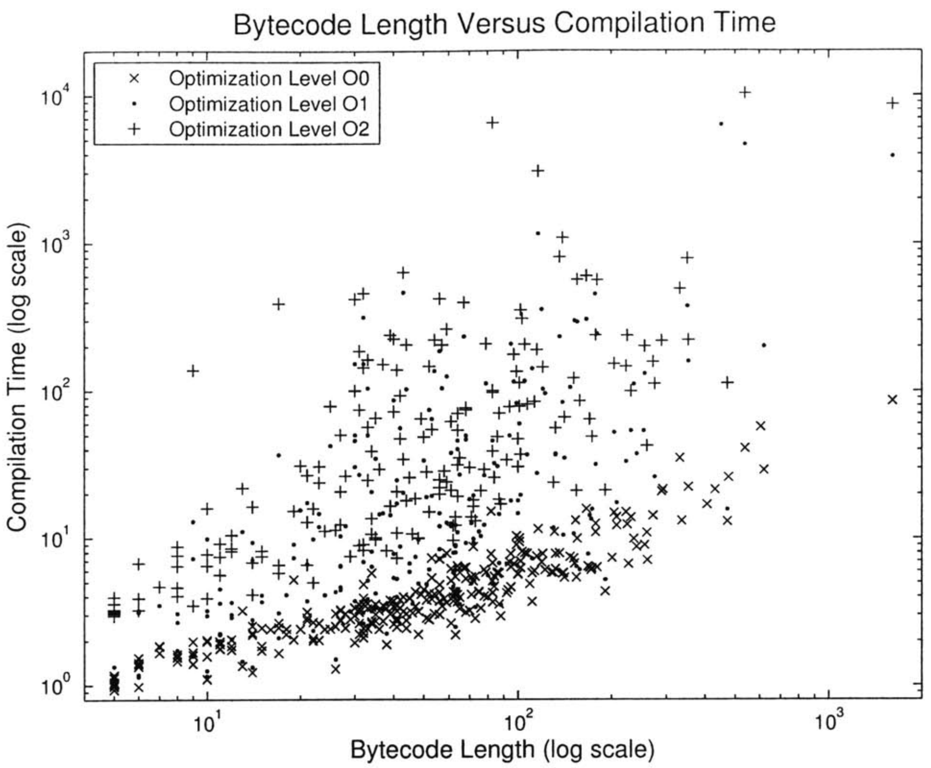


Figure 8-4: Bytecode length versus compilation time. This figure shows the relationship between compilation time and bytecode length on a logarithmic plot. For optimization level O0 the bytecode length is a good predictor of compile time, but for other optimization levels it is not the most informative feature. In particular, notice the large amount of scatter for optimization levels O1 and O2.

of compile time for optimization level O0 (marked with $\times$), it cannot adequately predict compile time for optimization levels O1 (marked with $\bullet$) or O2 (marked with $+$). The biggest reason why the O0 data are so linear is that the compiler does not aggressively inline methods at O0 (it only inlines small statically resolvable methods). Thus, the bytecode length accurately represents the "amount" of code that is compiled. The higher optimization levels aggressively inline, and thus inlining a single call can dramatically increase the size of the method under compilation. Thus, the multi-dimensional Ridge regressor we use finds the number of method calls (numInvoke) in the method under compilation extremely useful for predicting compilation time.

When we include more than one feature, our regressor's ability to predict compile

Index	Feature (x_i)
1	hasSynch
2	hasThrow
3	hasSwitch
4	sizeEstimate
5	numAllocation
6	numInvoke
7	numFieldReads
8	numFieldWrites
9	numArrayReads
10	numArrayWrites
11	numFwdBranches
12	numBkwdBranches
13	localVarWords
14	operandStackWords
15	bytecodeLength

Table 8.3: Feature Indices. These indices indicate the meaning of the value of each x_i in (8.1). Table 8.4 gives the coefficients β_i for each feature x_i.

Optimization Level	β_1	β_2	β_3	β_4
Level O0	0.5461	0.0	0.0	0.0
Level O1	0.6599	0.2804	0.0	−0.2405
Level O2	0.0	0.0	0.0	0.3573
Optimization Level	β_5	β_6	β_7	β_8
Level O0	0.237	0.2074	0.0344	−0.0304
Level O1	0.6565	0.9751	0.0	−0.1841
Level O2	0.222	0.5416	0.0	−0.2869
Optimization Level	β_9	β_{10}	β_{11}	β_{12}
Level O0	0.1313	0.0	0.0512	0.0832
Level O1	0.0	0.0	0.0	0.0
Level O2	0.0	0.0	0.0	0.3049
Optimization Level	β_{13}	β_{14}	β_{15}	β_0
Level O0	0.1671	−0.142	0.2025	0.3085
Level O1	−0.2062	0.0	0.5204	0.3624
Level O2	0.0	0.0	0.2745	0.7482

Table 8.4: Coefficients of Learned Predictors. These are the coefficients of the function given in (8.1) for each optimization level. The features corresponding to each β_i are given in Table 8.3.

time improves. As mentioned in Subsection 7.2.3 our regularized Ridge regressor naturally selects the best features from our feature set by driving the regularization parameter to zero for unimportant features. The form of the solutions is given by

$$\log(\mathit{compileTime} + 1) = \beta_0 + \sum_i \beta_i \cdot \log(x_i + 1). \tag{8.1}$$

The meanings of the x_i are given in Table 8.3, and the values of the β_i for each optimization level are given in Table 8.4.

We can see in Table 8.5 the prediction accuracy of the improved predictor obtained using Ridge Regression and the default predictor of Jikes RVM, based on bytecode length. The error for the improved predictor is based on 10-fold cross-validation, and therefore may slightly underestimate the true error. The error is generally smaller for the improved predictor than the default predictor. One exception is the relative error for O2. It is surprising that the default predictor has smaller relative error, but greater root mean square error, because by using a log transform, the regression minimized the relative error. However, looking at the percentage of data points that the predictors overestimate, which is a rough estimate of the bias of the predictor, we can see that the improved prediction is biased toward higher values, while the default is biased to-wards lower values. This may be due to the few large data points in our data set; although the log transform diminished the weight of these data points, they still pulled the regressor above the proper value.

Figures 8-5, 8-6, and 8-7 graphically depict the error of each compilation time predictor. In each plot, the heavy line indicates the actual compilation time. The data points have been sorted in increasing order of this value. For each data point, the improved prediction (marked with $+$) and the default prediction (marked with $\times$) are shown.[3]

[3]The default predictions for the ps benchmark are not shown, as mentioned previously.

		RMS Error	Absolute Relative Error	% Overestimate
Level O0	Default	12.1	0.92	18
	Improved	2.5	0.14	49
Level O1	Default	594.2	0.98	46
	Improved	404.3	0.83	48
Level O2	Default	1134.0	0.69	57
	Improved	961.0	0.86	41

Table 8.5: Prediction Error. This table gives the prediction error for our improved compilation time predictor and for the default Jikes RVM predictor, for each optimization level. The error for the improved predictor is based on 10-fold cross-validation. The root mean square (RMS) error is the square root of the mean of each error squared. The absolute relative error is the mean of the absolute values of each error divided by the correct value. The percent overestimate is the percentage of the predictions that are above the correct value, and indicates bias in the predictor.

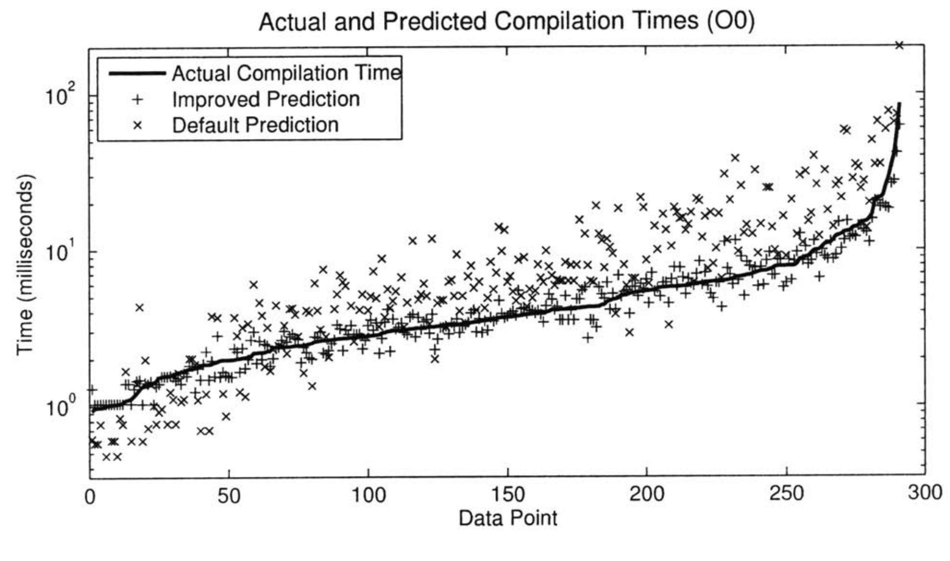


Figure 8-5: Prediction Error, Optimization Level O0. This figure graphically shows the predictions for our improved compilation time predictor and for the default Jikes RVM predictor, for optimization level O0. The solid line shows the actual value.

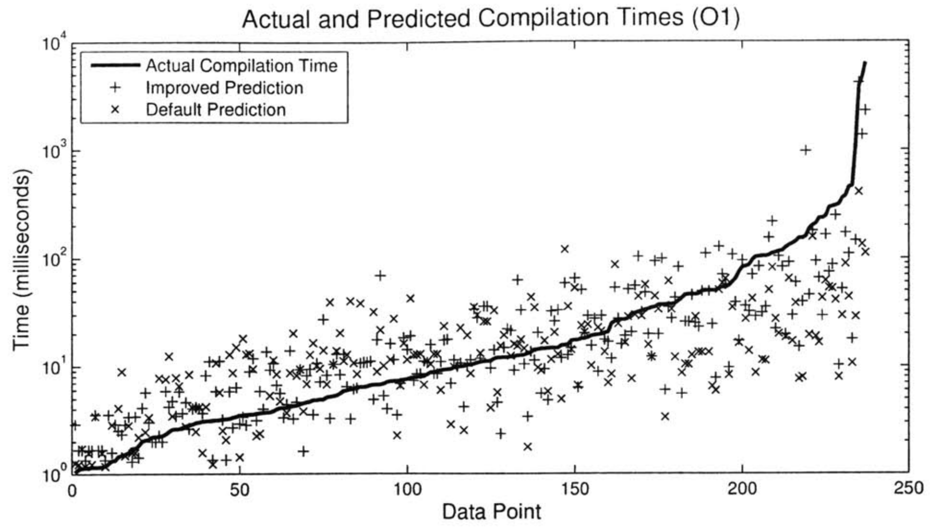


Figure 8-6: Prediction Error, Optimization Level O1. This figure graphically shows the predictions for our improved compilation time predictor and for the default Jikes RVM predictor, for optimization level O1. The solid line shows the actual value.

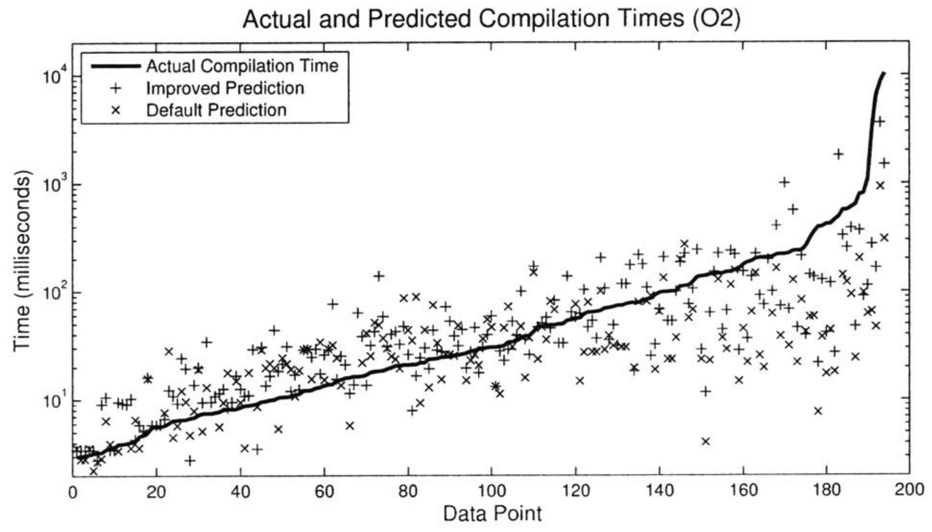


Figure 8-7: Prediction Error, Optimization Level O2. This figure graphically shows the predictions for our improved compilation time predictor and for the default Jikes RVM predictor, for optimization level O2. The solid line shows the actual value.

8.5 Performance Gain using Compilation Time Predictor

We conclude this section by showing the speedups attained by using the Ridge regressor to predict compile times. Figure 8-8 shows that by simply repeatedly running the eight benchmarks in the graph, our system can converge upon better solution than the default heuristic (4% overall for the benchmarks in the DaCapo benchmark suite [18]). In some cases, the improvement is dramatic. For instance, `bloat` was improved by 14%, and `hsqldb` by 10%. There are three cases in which our predictor slows down the VM. The reason for the slowdowns is simple: we are still operating under the assumptions stated in Section 2.3. Namely, we still assume constant runtime improvements from lower to higher levels of optimization, and we still assume that a method's remaining runtime will be equal to its current runtime. These imperfect assumptions can lead to suboptimal performance even when the compile time is perfectly predicted. Future work will expand on these results by incorporating predictions for runtime benefits.

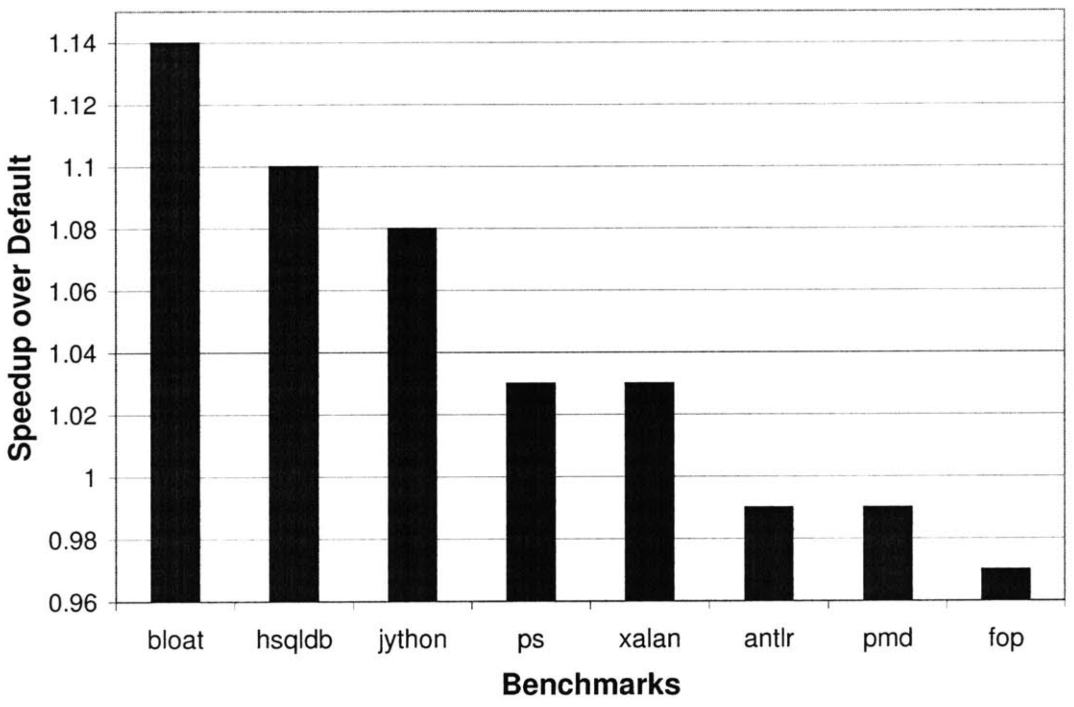


Figure 8-8: Improved Performance with Accurate Prediction. The improved prediction accuracy of the Ridge regressor usually improves the total overall runtime of the VM. Our system reduces the total overall runtime on five of the eight benchmarks in the DaCapo benchmark suite. The geometric mean on DaCapo is a 4% overall improvement.

Chapter 9

Related Work

Collaborative compilation draws from several areas of compiler research, including adaptive compilation, automated heuristic generation, instrumentation, and remote feedback systems. Collaborative compilation has similarities with other research in each of these areas, but it is a combination of all these areas that makes it very unique in the field.

9.1 Adaptive Compilation

Arnold *et al.* [3] give a survey of adaptive, feedback-directed, and profile-directed compilation techniques for virtual machines. (We use the term "adaptive compilation" to mean what they term "selective compilation" in addition to their definition of adaptive compilation.) Collaborative compilation has aspects of adaptive compilation and feedback-directed compilation, but most closely fits a profile-directed compilation technique.

9.1.1 Profile-Directed Compilation

Although many researchers have used profile-directed compilation techniques, we will focus on those most related to collaborative compilation. Although Wall [45] claims that estimated profiles requiring low overhead are far inferior to full and intrusive profiles, all of the projects cited here used sampling techniques, as does collaborative compilation.

Pan and Eigenmann [37] used profiling techniques to tune compilers by comparing versions compiled in different ways. In comparing their work to expensive "at-the-factory" tuning techniques, they claimed low overhead, but in comparison to other profile-directed techniques, the overhead is quite high.

Long and O'Boyle [30] used profiling techniques to find the best loop transformations. Like collaborative compilation, data is collected as programs are compiled and run, leading to new example programs in the knowledge base. Although the authors apply the machine learning process only to transformations for loops and arrays, they claim that using Pugh's Unified Transformation Framework, many other optimizations are possible. They require that all potential transformations be tested initially, for example, when the compiler is ported to a new platform.

Arnold, Welc, and Rajan [5] used low-overhead profiling techniques to record the running times of methods in a Java virtual machine. These profiles were used in subsequent runs to estimate the future running time of the methods, improving the choices made by the adaptive system. These estimates are the running-time analog of our compile-time predictions, discussed in Section 8.4. They gave the option of performing analysis of the profiles either offline or incrementally at the program exit. Their system can dynamically adapt to changes in program behavior or inputs across runs.

All of these profile-directed approaches use local profiling to realize local performance improvement. This results in much redundant computation as the same basic information is gathered at each site. In contrast, collaborative compilation has the benefits of profile-directed optimization in that it tunes the compiler for the applications that are being run, but also has the benefit of "at-the-factory" training, in which training is required only once for all users.

9.1.2 Feedback-Directed Optimization

Feedback-directed optimization also relies on profiling techniques, but the data is applied during the same run. Thus, although anomalies of the current program run are detrimental to profile-directed optimization, feedback-directed optimization can exploit these anomalies. Dynamic Feedback is a good example of this approach, by Diniz and Rinard [19]. Similar to collaborative compilation, their system compiles several versions of a code section using different optimization techniques. They then profile for one phase, measuring each version to find the one with the best performance, and then use the best version for a production phase. They alternate between these two phases to detect changes in program behavior. Lau *et al.* [26] have implemented a similar technique.

Voss and Eigenmann [44] allow feedback-directed optimization to be used more extensively by offloading the recompilation required to a compilation server.

9.2 Machine Learning for Compiler Optimization

Recently there has been a great deal of research on applying machine learning and artificial intelligence techniques to help solve difficult compiler problems. Collaborative compilation allows for the use of machine learning, but is general enough for other approaches as well. We discuss the many techniques that other researchers have used that fit naturally into the collaborative compilation framework.

Of particular note, this project was highly influenced by previous work involving Stephenson, Amarasinghe, and others [42, 43, 38].

9.2.1 Supervised Learning

Among those using supervised learning, there were a variety of applications. Moss *et al.* [33] improved instruction scheduling using various supervised learning techniques, including neural nets and decision trees. Calder *et al.* [7] improved branch prediction using neural nets and decision trees. Monsifrot, Bodin, and Quiniou [32] worked on loop unrolling using a decision tree algorithm. Long and O'Boyle [30] concentrated on loops and arrays using nearest-neighbors. Stephenson *et al.* [42] used nearest-neighbors to predict the optimal unroll factor out of eight possibilities. Cavazos and Moss [8] used rule-set induction to decide whether or not to perform the instruction scheduling phase of the compiler, and with O'Boyle [9] for choosing between two algorithms for register allocation, linear scan and graph-coloring.

All of these projects used only machine learning to arrive at a heuristic for the target optimization. In contrast, Loh and Henry [29] used hand-crafted heuristics, but chose which heuristic to use through machine learning, creating an ensemble predictor.

9.2.2 Directed Search Strategies

Many authors have used search strategies, most commonly genetic programming, to find the best compilation heuristics, similar to our strategy of policy search. Each of the following projects have used genetic programming as a strategy to search the space of possible heuristics. Beaty [6] applied this technique to instruction scheduling. Stephenson *et al.* [43] learned priority functions for hyperblock formation, register allocation, and data prefetching. Puppin *et al.* [38] learned the best sequence for instruction scheduling passes in convergent scheduling. Similarly, Kulkarni *et al.* [25] used genetic programming to optimize the sequence of optimization passes. Finally, Cavazos and O'Boyle [10] applied this technique to learn inlining parameters in a Java virtual machine.

Cooper and collaborators [15, 16, 14, 17] have been quite prolific in researching the search space of compiler heuristics. In [14], Cooper *et al.* explore the search space of sequences of optimization passes, showing that the space has many local minima and maxima. Cooper, Subramanian, and Torczon [16] searched for optimal compilation sequences within this space using biased random sampling and genetic programming. Cooper, Scheilke, and Subramanian [15] also searched the space of compilation sequences, but their goal was to reduce code size rather than decrease running time. Cooper and Waterman [17] used a hill-climbing approach to find the best parameter for blocking size in matrix operations.

9.3 Instrumentation and Profiling

Our method of sampling (Section 4.2.1) is somewhat similar to that of Arnold *et al.* [4], in that we periodically sample a separate copy of the code that is instrumented.

Our method of relative measurement (Subsection 4.2.2) is somewhat similar to those of Pan and Eigenmann [37]. Our goals are also somewhat similar; they mention the long tuning process of the work in Section 9.2. However, their approach is to reduce the time for offline tuning, while our approach is to use very low-overhead online tuning. Although they make dramatic improvements in reducing the time required for the "at-the-factory" training phase of previous work, their approach still requires a lengthy training process.

9.4 Automatic Remote Feedback Systems

Liblit *et al.* have pioneered automatic remote feedback systems for bug isolation [28, 27]. Like collaborative compilation, these systems gather data from a user community. This information is used to help application developers locate and fix bugs quickly. They address privacy and security concerns in their system in [27].

9.5 Client-Server Design for Compilation

As mentioned in Chapter 6, a client-server design is rarely used for compilation. However, three projects showed good results using this design; all used a server to compile code for clients. Onodera [36] used a local compilation server or daemon to avoid recompiling C header files for repeated or mass compilation. Voss and Eigenmann [44] used an optimization server to eliminate compilation time for dynamic optimization. The service was designed to run either on a remote system or on a multiprocessor. A compilation server is a natural design for embedded systems, where resource constraints limit runtime compilation. Newsome and Watson [34] dramatically decrease the necessary code size for natively-compiled Java programs by eliminating the need for an interpreter or JIT compiler for dynamically-loaded code. The performance is dramatically superior to an interpretation strategy and in most cases faster than JIT compilation.

Chapter 10

Conclusion

Collaborative compilation makes profile-directed optimizations general, ubiquitous, timely and efficient. In contrast to the traditional approach, collaborative compilation has very low overhead and gives performance improvements with minimal training time. It works for many compiler problems and can be extended easily to tackle these problems.

Collaborative compilation can give performance improvements quickly because it collects information from many clients simultaneously. With this massively parallel data collection technique, each client does a minimal amount of work to realize the gains from the sum of the community. Because of its low overhead, users will be willing to allow data to be collected, helping to make collaborative compilation ubiquitous.

We have demonstrated collaborative compilation giving performance improvements by tuning three compiler problems. Modifying the compiler to perform the tuning was easy, as the collaborative infrastructure was already in place. This extensibility makes collaborative compilation general enough to handle many compiler problems, and aids in its adoption by compiler developers.

In sum, collaborative compilation gives most of the performance improvements of program-specific profile-directed optimization with negligible overhead. It also provides immediate performance improvement on unseen programs by using a shared knowledge base.

10.1 Contributions

This work provides many contributions to the field of compilers and program optimization. We list a few of these contributions here.

- We have laid the conceptual groundwork that makes collaborative compilation possible, from the general design to specific problems of measurement.
- We have implemented a prototype collaborative system. Other researchers and compiler developers can draw inspiration from this prototype and extend it.
- We have used collaborative compilation to tune the Jikes RVM compiler, providing three examples of collaborative compilation improving performance by tuning the adaptive system, the inlining heuristic, and the compilation time predictor.
- We have provided greater insight into the factors that contribute to compilation time. We have shown a nonlinear relationship that predicts compilation time well. Not only will this help improve compilation time predictions, it may also result in changes to the compiler to account for the previously unrecognized factors.

We look forward to others using our contributions and building on our results.

10.2 Future Work

As with any novel concept, there were far more ideas for extending collaborative compilation than there was time to implement them. We also raised many questions that we did not have time to answer definitively. This section invites future research on these topics.

10.2.1 Additional Optimizations

We have implemented collaborative compilation for several compiler problems. In the future, we hope to add additional optimizations and collaborate with others to do so. As we add optimizations, the improvements for one may interact with another, causing unforeseen results. It is our hope that with several optimizations, each will evolve together to create greater symbiosis, improving overall performance. However, it is also possible that performance will oscillate or change chaotically.

10.2.2 Privacy and Security

The basic privacy and security safeguards of collaborative compilation are described in Section 6.4. However, there are many interesting questions in this regard that we have not explored.

Since the collaborative server controls some aspects of the performance of the client's code, it is possible that a malevolent server could cause very poor performance. It seems likely that safeguards on the client could prevent such an attack, but it is an open question how that could be engineered and whether these safeguards would prevent the system from achieving optimal results. Likewise, a malevolent client could cause the entire community to experience degraded performance by submitting bogus data. Cryptographic signatures seem like a viable solution to these two problems.

As mentioned in Section 6.4, it is possible that the feature vectors contained in a program act as a "fingerprint" of that program. An untrusted server may be able to analyze the distribution of the feature vectors sent by a particular client over a period of time to predict the programs that the client runs. Our intuition leads us to believe this is possible—consider the accuracy with which we predicted compilation time in Section 8.4. Taking into consideration the performance data, it is likely that at least the size of the input, if not more information, could be deciphered by the server. However, these are also open research questions.

We claimed that the information leakage through policy search would be small. The compilation policy has full access to the features of the program, and also a way to signal the server, albeit limited, by causing very poor performance. It would be interesting to know how how small this information leak is.

A more interesting question is how much information can be gleaned from the heuristics published by a collaborative server. We do not know how much, if any, information about the clients can be gained by analyzing the heuristic arrived at as a result of the clients' data. For example, suppose a company creates a collaborative server for all employees to use, and publicly publishes its heuristics. If the employees mostly use a single application, there may be peculiarities in the published heuristics due to this program.

10.2.3 Phase Behavior

In Subsection 4.1.3, we describe phases in terms of the problems that phase changes can cause for collaborative measurement. However, if the system can properly detect phases, collaborative compilation is an ideal infrastructure in which to exploit the differences between phases to improve performance.

10.2.4 Community Dynamics

We have shown collaborative compilation to be successful in simulation, where each user contributed information about his or her application of interest. However, if a collaborative user is "hanging out with the wrong crowd," that is, if he or she subscribe to a community that generally has different interests, the system may not be advantageous for this user. Indeed, our system degrades performance for some benchmarks (see Figure 8-8). Additional research into the community dynamics of the system may shed some light on this behavior.

Bibliography

[1] *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Monterey, California, December 2000.

[3] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, February 2005.

[4] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 168–179, New York, NY, USA, 2001. ACM Press.

[5] Matthew Arnold, Adam Welc, and V.T. Rajan. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *Proceedings of the SIGPLAN '05 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[6] Steven J. Beaty. Genetic Algorithms and Instruction Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, November 1991.

[7] Brad Calder, Dirk Grunwald ad Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. In *ACM Transactions on Programming Languages and Systems (TOPLAS-19)*, volume 19, 1997.

[8] John Cavazos and Eliot Moss. Inducing Heuristics to Decide Whether to Schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004*. ACM, 2004.

[9] John Cavazos, Eliot B. Moss, and Michael F.P. O'Boyle. Hybrid Optimizations: Which Optimization Algorithm to Use? In *15th International Conference on Compiler Construction (CC 2006)*, Vienna, Austria, March 2006.

[10] John Cavazos and Michael F.P. O'Boyle. Automatic Tuning of Inlining Heuristics. In *11th International Workshop on Compilers for Parallel Computers*, January 2006.

[11] Tony F. Chan and John Gregg Lewis. Computing standard deviations: Accuracy. *Communications of the ACM*, 22(9):526–531, September 1979.

[12] Yin Chan, Ashok Sudarsanam, and Andrew Wolfe. The effect of compiler-flag tuning on spec benchmark performance. *ACM SIGARCH Computer Architecture News*, 22(4):60–70, September 1994.

[13] Commons-Math: The Jakarta Mathematics Library. *http://jakarta.apache.org/commons/math/*.

[14] Keith Cooper, Alexander Grosul, Timothy J. Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms. In *Los Alamos Computer Science Institute Symposium*, 2003.

[15] Keith Cooper, Philip Scheilke, and Devika Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In *Languages, Compilers, Tools for Embedded Systems*, pages 1–9, 1999.

[16] Keith Cooper, Devika Subramanian, and Linda Torczon. Adaptive Optimizing Compilers for the 21^{st} Century. In *Los Alamos Computer Science Institute Symposium*, 2001.

[17] Keith Cooper and Todd Waterman. Investigating Adaptive Compilation using the MIPSpro Compiler. In *Los Alamos Computer Science Institute Symposium*, 2003.

[18] The DaCapo Benchmark Suite. *http://ali-www.cs.umass.edu/DaCapo/gcbm.html.*

[19] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–84, 1997.

[20] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer, New York, New York, July 2001.

[21] Java Grande. *http://www.epcc.ed.ac.uk/javagrande/.*

[22] Daniel A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 107–116, New York, NY, USA, 2005. ACM Press.

[23] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[24] JUnit.org. *http://www.junit.org/.*

[25] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding

effective optimization phase sequences. In *In Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*. ACM, 2003.

[26] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online Performance Auditing: Using Hot Optimizations Without Getting Burned. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 conference on Programming Language Design and Implementation*, 2006.

[27] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* [1].

[28] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, Chicago, Illinois, June 12–15 2005.

[29] G.H. Loh and D.S. Henry. Applying machine learning for ensemble branch predictors. *Developments in Applied Artificial Intelligence*, pages 264–74, 2002.

[30] Shun Long and Michael O'Boyle. Adaptive java optimisation using instance-based learning. In *International Conference on Supercomputing*, 2004.

[31] Nikki Mirghafori, Margret Jacoby, and David Patterson. Truth in spec benchmarks. *ACM SIGARCH Computer Architecture News*, 23(5):34–42, December 1995.

[32] Antoine Monsifrot, François Bodin, and René Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 41–50, 2002.

[33] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanović, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. In *Proceedings of Neural Information Processing Systems*, 1997.

[34] Matt Newsome and Des Watson. Proxy compilation of dynamically loaded java classes with mojo. In *LCTES/SCOPES '02: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 204–212, New York, NY, USA, 2002. ACM Press.

[35] NIST/SEMATECH. *NIST/SEMATECH e-Handbook of Statistical Methods*, May 2006.

[36] Tamiya Onodera. Reducing compilation time by a compilation server. *Software - Practice and Experience*, 23(5):477–485, 1993.

[37] Zhelong Pan and Rudolf Eigenmann. Rating compiler optimizations for automatic performance tuning. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 14, Washington, DC, USA, 2004. IEEE Computer Society.

[38] Diego Puppin, Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Adapting Convergent Scheduling Using Machine Learning. In *Proceedings of the '03 Workshop on Languages and Compilers for Parallel Computing*, College Station, TX, 2003.

[39] Sheldon M. Ross. *A First Course in Probability.* Prentice-Hall, Upper Saddle River, New Jersey, sixth edition, 2002.

[40] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.

[41] SPEC.org. *http://www.spec.org.*

[42] Mark Stephenson and Saman Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *International Symposium on Code Generation and Optimization*, San Jose, California, March 2005.

[43] Mark Stephenson, Martin Martin, Una-May O'Reilly, and Saman Amarasinghe. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation* [1].

[44] Michael J. Voss and Rudolf Eigenmann. A framework for remote dynamic program optimization. In *DYNAMO '00: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 32–40, New York, NY, USA, 2000. ACM Press.

[45] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 59–70, Toronto, Ontario, Canada, June 1991.

[46] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, second edition, 2005.