Machine Learning Algorithms for Choosing Compiler Heuristics

by

Gennady G. Pekhimenko

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Machine Learning Algorithms for

Choosing Compiler Heuristics

Gennady G. Pekhimenko

Master of Science

Graduate Department of Computer Science

University of Toronto

2008

During the last several decades compiler developers have invented a set of powerful heuristics to deal with the complexity of the algorithms they have to use. However, this led to a new problem of finding the best values for every heuristic. This paper describes how machine learning techniques, such as logistic regression, can be used to build a framework for the automatic tuning of compiler heuristics. In this paper we were focused on decreasing the compile time for the static commercial compiler called TPO (Toronto Portable Optimizer) while preserving the execution time. Nevertheless, our techniques can also be used for decreasing the execution time and in dynamic behavior. Our experiments showed that we can speedup the compile process by at least a factor of two with almost the same generated code quality on the SPEC2000 benchmark suite, and that our logistic classifier achieves the same prediction quality for non-SPEC benchmarks.

Acknowledgements

I would like to thank my supervisor, Angela Demke Brown, for her guidance during my research which made possible this work to appear. I am also grateful to Mathew Zaleski for his comments throughout my work.

The implementation of the ideas in this thesis would not have been possible without the help of the TPO compiler team at the IBM Toronto Lab, and especially Arie Tal and Raul Silvera.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

What we call "Progress" is the exchange of one nuisance for another nuisance.

— Havelock Ellis

Compiler writers need to implement many optimizations that are NP-hard problems, but still need to be solved efficiently. In order to address this problem a number of powerful heuristics were invented. However, this led to a new problem of finding the optimal parameter value for every heuristic. The solution for this task is still a mixture of art and experience and often needs a great amount of human time and effort. There are several reasons that finding the optimal values is difficult. The first problem is the highly non-linear interaction between optimizations, such that a change in one transformation needs unpredictable adjustments to others. The second problem is that the number of these heuristics can be huge, and thus it is impossible to check the search space completely.

Hence, the idea of making this process as automatic as possible seems very natural. One of the ways to do this efficiently is using machine learning (ML) algorithms, which can be powerful tools for searching for optimal solutions in a high dimensional space. Previous work showed several methods that can be applied for such a task. Among them are *nearest neighbors* (NearN) [16, 1], *logistic regression* [6, 5], *neural networks* (NN) [4], *genetic programming* (GP) [17], and *support vector machines* (SVM) [16]. Thus, if we can choose a group of valuable transformations from our list of compiler optimizations, it is possible to make a tool for finding the optimal values for heuristics inside these transformations using ML techniques, thereby improving overall performance.

Most of the previous works were focused on decreasing execution time or total time (in the dynamic case), but for commercial static compilers the compilation time can also be an issue. Most of them have a significant number of transformations that can use all the information about the target platform, programming language, and even hints about the specific application. However, it is hard to predict when and where these transformations are needed, so in most cases compiler writers apply all of them to every piece of code. As the result, the compilation time can become a significant problem. Using different levels of optimizations (e.g. -O0, -O2, -O3) with fixed sets of transformation provide some flexibility for the customers to trade off compile time and code quality. But there are still many cases when you need the quickest code you can get with reasonable compilation time (e.g. OS or Web Server compilation). Hence, you need a way to predict what transformations are needed for a particular program and apply only this list of transformations.

To distinguish different programs you need a way to characterize them. We suppose that the whole program level is not an appropriate level, because a program contains many functions with different characteristics. Applying the same set of transformations to all of them is unlikely to give the best results. Similarly, when "features" are aggregated across a whole program, two programs may appear similar when in fact they should use different transformations for best behavior. Hence, we need a method-based description for our purposes. After we have a method description we can use different techniques to predict the set of transformations that is the best for this method. We decided to apply a machine learning technique called *logistic regression* to predict the best set of transformations between two levels inside the IBM Toronto Portable Optimizer (TPO).

TPO is based on IBM's xl compiler infrastructure, supports several languages (e.g. C/C++, Fortran), and generates highly optimized code. Two levels were chosen: one with acceptable execution time (*-O3 -qhot*) and one with the desired compilation time (*-O3*). There are around 50 transformations between these levels that require a lot of compilation time, but in most cases do not improve execution time. We gathered training data by disabling transformations for every hot function for a set of SPEC2000 benchmarks. Then, we used this data to predict the transformation set for every function in SPEC2000 benchmarks as well as for other benchmarks. We showed that it is possible to decrease the compilation time at least 2.46 times for SPEC2000, coming close to the compilation time at *-O3* level. At the same time we did not increase, but even improve the average execution time by 2% (5 tests from SPEC2000 significantly improved). Some previous works achieved a similar speedup when the execution time was their primary goal [16, 17].

In order to achieve our goals we created a framework that consists of additional options and opportunities inside TPO, a set of perl scripts to get training data and find the best set of transformations, and the ML algorithm implemented as Matlab([11]) functions. This framework is flexible enough to add new code features, new transformations to analyze and different types of heuristics (not necessarily binary). We expect that with minor changes this framework can be used to improve the execution time as the primary goal and it can also be used for dynamic compilers or JIT (just-in-time) optimizers to control their decisions automatically.

The remainder of this paper is organized as follows. The next chapter introduces our motivation in more detail. Chapter 3 describes the main ideas of our approach including feature extraction, heuristic modifiers, gathering training data, and the learning process. Chapter 4 discusses the methodology that we used and the infrastructure that was created to achieve the project goals. It also describes the measurements we did to show the result of our technique. Chapter 5 presents and discusses the results. Chapter 6 relates our work to previous work on similar topics. We conclude and describe the potential of this work in Chapter 7.

Chapter 2

Motivation

Some men have thousands of reasons why they cannot do what they want to; all they need is one reason why they can.

— Willis Whitney

This chapter provides more detail to motivate our project goal and the means (e.g. machine learning, the industry compiler) we chose to achieve them. There are also several more goals that can be reached using the framework we have implemented, and we describe the usage possibilities for this framework and the reasons why certain changes and additions are needed in the existing compiler framework.

2.1 Why Compile Time?

As we mentioned in Chapter 1, decreasing the compilation time is our primary goal for this project. However, it is easy to argue that compiling is only a one-time process and then you will use efficiently generated code frequently. For most simple cases this is fine, but there are big projects where this can be a serious problem. As an example, we can consider the case of compiling a real industry Operating System (OS). It has several critical sections of code that should be compiled with the highest optimization level, while the rest do not need to be optimized so extensively. Compiling with the traditional approach may lead to a compile time of more than one day using TPO. And this is not the whole problem. It is a common case when there are bugs either during the compile process or later after the execution. For debugging purposes you will probably need to recompile the OS at least several times, and a delay while waiting for the code to compile slows down this process significantly.

Our approach gives an opportunity to avoid the delay by using the full "power" of the optimizer only if it is needed. This benefit is not a serious advantage for a user with a simple application to compile and run, but it is certainly interesting for industry or serious scientific projects. It is also possible to improve the execution time in some cases, because the transformations can sometimes do "bad" things with your code. Preventing them from applying to a code can sometimes give a speedup in the execution time.

2.2 Why Machine Learning and TPO?

Another question that may arise is why we apply the ML technique and not something else. We are not going to prove that ML is the best possible approach, but the nature of our problem - predicting something about the code using its characteristics, appears to be a suitable application for one of the artificial intelligence (AI) methods. The expert systems (ES) were the second possibility that we kept in mind, but the complexity of retrieving the "expert opinion" about the transformations stopped us from using it. Instead we chose the ML technique called logistic regression, which will be described in Section 3.6.2 in more detail. It is possible to use our framework to retrieve some knowledge about the optimizer that can be used as a part of the ES, but we currently do not put any efforts to investigate this possibility.

At the starting point of our project we were thinking about what compiler to choose: a commercial one or an open source alternative. Except for the traditional advantage of the open source projects (i.e. freely available code) these compilers are usually less complicated. There was probably less effort in finding the best values for their optimization heuristics, which gives us a better chance to show good results. However, these projects usually do not have enough resources to make their code very stable, and our changes in the heuristic values can cause an unpredictable number of bugs that will not be fixed quickly. Previous works showed that all these bugs may happen, for example the unroll factor limitation in [16].

The second reason for choosing a commercial compiler was the desire to get benefit for something that has really good performance while executing, and is so complex that it is almost impossible to do this work manually. The final reason was the existence of the Heuristic Context approach [18], which significantly decreases the changes we need to make inside TPO. We describe the Heuristic Context approach in more detail in Section 3.3.

2.3 Initial Experiment

In order to verify that we have possibilities for decreasing the compile time we conducted the following simple initial experiment for TPO. We measured the compile time for two levels, *-O3* and *-qhot -O3*, to see whether there is enough difference to start optimizing. The results were promising, showing more than 3 times difference for most of the twenty-one SPEC2000 [8] benchmarks, as shown in Figure 2.1. These results were used as the baseline for the future measurements.

2.4 Different Ways of Tuning

The process of searching for the optimal values of the transformation heuristics is usually called the *tuning process*. We can apply tuning in different ways - both for individual

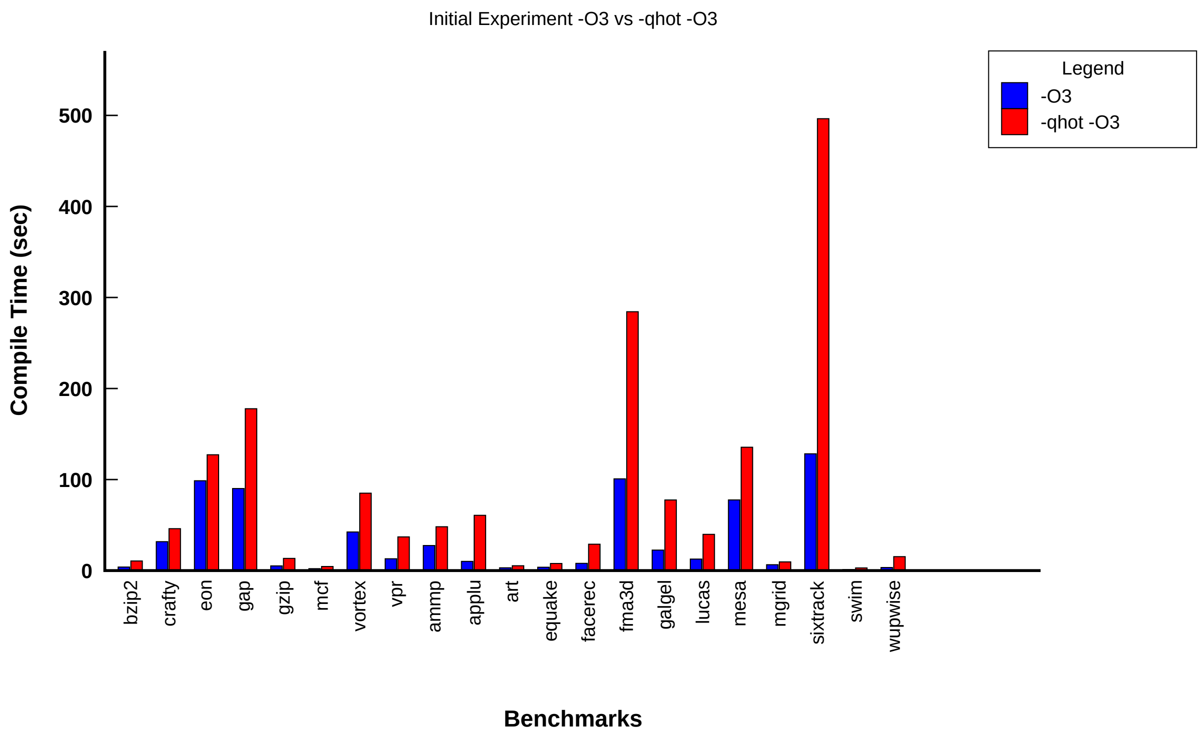

Figure 2.1: Comparison of the compile time for two optimization levels (-O3 and -qhot -O3) on SPEC2000 benchmarks.

optimizations and for any group optimizations in which we are interested. The method we choose will influence most of our subsequent design decisions. The problem of tuning certain optimizations by ML was discussed early in several works. Examples include: hyperblock formation, register allocation and prefetching [17]; unroll factors [16]; and branch prediction [4]. These papers have several promising results, but they did not try to analyze any group of optimizations as a whole object for learning process. However, there are a lot of dependencies between different optimizations, and, hence, to obtain the best performance level we should try to apply learning for certain groups of optimizations or even for the whole set of transformations.

As a good simple example, consider a pair of optimizations called in-line expansion and loop unrolling [12]. In-line expansion is based on the idea of avoiding the call overhead by replacing it with the function body. Loop unrolling is a well-known transformation in which the loop body is replicated a number of times. It reduces overhead by decreasing the number of branch operations, because the backward branch is needed only after executing the whole unrolled loop body. Both of them increase the code size and, if they are not limited by some parameter, can be the source of instruction cache misses. If we tune them separately, first, we will find a limitation for the first optimization, say in-line expansion. Later, we can try to tune the unrolling and the increase of the code size may be a problem for some applications. However, if we apply in-line expansion less aggressively we might be able to achieve greater speedup by using more unrolling.

A somewhat similar problem arises when you are trying to find the best order for a group of optimizations [6]. As a side effect, we can find interesting dependencies between optimizations that can lead to a redesign in our optimizer. We are not focusing on the order of optimizations/transformations in this paper, and assume that the compiler writers chose the optimal or at least close to the optimal order for transformations. In general, it is also an interesting issue to be solved and ML techniques could be applicable again.

In order to create a framework that can be flexible enough to solve all mentioned tasks, we make it possible to tune any specified set of heuristics for every transformation and to disable/enable every optimization. While most of our experiments were conducted with the goal of finding the smallest set of transformations with the acceptable execution time, it is still possible to tune just a single heuristic value.

2.5 Technological Process

The described capabilities of this framework can be used not only for existing optimizations, but for every new transformation that is going to be added inside the optimizer. This approach will significantly decrease manual tuning, and, hence, will save human work time for other tasks. At the same time, when you add something new inside the optimizer, it may be necessary to re-tune for the previous transformations. This can be hard to do manually (even for experienced compiler writers), and can be the source of performance inefficiency. Ideally we should use the framework for every new optimization, and it can become a part of existing technological process for the compiler development process. This will decrease the time for making a new compiler and make it more stable and predictable.

A good description of characteristics that should be present in the modern compiler to deliver a satisfactory level of performance, to be flexible enough to deal with new hardware, and to avoid fixed or unclear heuristics is presented in *Building a Practical Iterative Interactive Compiler* [10]. The authors mentioned a set of characteristics that should be present in a good compiler framework. Most of them are present in our approach: reuses for the compiler program analysis, a simple and unified mechanism to obtain information about compiler decisions, transparency for user (at high level), fine-grain (function level) and only legal transformation for a given application. Unfortunately, TPO did not have all of these capabilities at the beginning, and, hence, it was necessary to expend some engineering efforts to extend the current optimizer.

2.6 Customer Application

It is a very common case when certain customers need better performance for their applications. Compiler writers usually choose their heuristic values based on the performance results for a set of well-known benchmarks, like SPEC benchmarks. Hence, the resulting parameters may be good only in general; in most cases reducing the geometrical mean of the execution time on the whole set of tests is the primary goal. But we can do much better in most cases for specific applications by changing the aggressiveness of our transformations. You can use compiler writers to do this routine job, and these people will be then a ”critical resource” if you have a lot of customers. Or you can use a ML tool/framework that will try to do this job automatically. In many cases such an automated tool can be enough, but it can be the case when you need to rewrite some optimizations or add new ones to increase this application's performance, making additional human work unavoidable.

2.7 Bug Finding

In every complex software tool there are a certain number of bugs. A compiler is not an exception and even industrial releases have hundreds of bugs. When you are changing parameters for one optimization inside your optimizer you will change the input to other phases, which can easily be the cause of bugs [17]. So, if you automatically execute this tool for different parameters, you can find different bugs that were in a compiler before, but were not tested with the appropriate input. In some cases it can be very useful to run optimization in ”stress” mode, for example for register allocation with a small number of available registers, the spill/fill technique can be tested better than in the general case. We can also find performance problems/bugs when disabling optimization makes the code faster. In our experiments we found both execution (gcc, perlbmk) and performance bugs.

In this chapter we described the reasons why our current project is interesting and what benefits it can provide. We conducted the initial experiment that showed the possibilities for decreasing the compile time. The next chapter will present the core of our work, including the description of all approaches and techniques we used in this research.

Chapter 3

Approach and Methodology

The beginning of knowledge is the discovery of something we do not understand.

— Frank Herbert

In this chapter we give a description of how we use the ML technique called logistic regression to predict a good set of compiler heuristics. The first section gives a brief overview of our approach. The next three sections describe the preparation phases that are needed to collect the training data, described in Section 3.5. Then, we describe both the learning and the deployment processes with the appropriate examples and algorithm overviews.

3.1 Overview

The whole process can be separated into four phases that do not intersect: data preparation, gathering training data, learning, and deployment. Data preparation itself consists of feature extraction, modifying heuristics and choosing the transformation set. Features describe characteristics for each function that may be relevant to particular optimizations; if they are chosen well, they can be used to predict the best set of transformations for a new function. That is why choosing the right set of features is one of the key points in this research. The Context Heuristic Modifiers [18], which were previously implemented in TPO, give a quick and easy way to change the current heuristic values. The final key point is choosing the right set of transformations to search through, because every additional transformation increases the search space. But at the same time, if we lose a valuable transformation it could prevent us from getting interesting results.

We gather training data in two steps. First, we get all hot functions for every single test using the profile tool called *tprof.* Then, we run every single test with different set of heuristics for all its hot functions. One of the main problems is how to make an efficient search over such a huge search space that increases exponentially with both the number of heuristic values and the number of hot functions. Details are provided in Section 3.5.

The main characteristic of the learning phase in our approach is that it can be done offline, just like the training phase. We do not need to have any significant overhead during the compile time because of the time-consuming learning. Instead, we perform learning beforehand using the training data and, then, we can use the learned parameters together with the features (which are easy to compute) to predict the best set of transformations for a new function. This makes it possible to use any learning algorithm we want with the same straightforward deployment implementation.

3.2 Feature Extraction

As we mentioned above, we need a way to describe every method/function if we want to predict something valuable about it. To be consistent with ML techniques [3] we called these characteristics *features* and put several constraints on them (more or less formal). The first constraint is to be a ”good” function description and have a higher chance to be different for the methods that are ”different” and be the same or close to the ones that seem to be ”similar”. There is no point in having a feature that is the same for all tests or has no connection with a decision of whether to apply any of the transformations. It is hard to make this more formal and we mostly reuse the characteristics that were already analyzed inside the compiler to make decisions about transformations. The second constraint is to be fast to compute, because we will have the overhead of extracting features for every method that is compiled. Complex features can be the reason for serious increase in the compilation time and this can kill the main goal of our project, which is decreasing the compilation time.

The set of features we used can be separated into two major categories: general and loop-based. The general features are the ones that characterize a method as to the number and/or percentage of a certain type of instruction (e.g. the number of loads or the percentage of branches). This group is listed in Table 3.1(a). These features are similar to the features used by other authors [6, 1]. The second group consists of the features that are either loop characteristics or need the loop tree to be computed (Table 3.1(b)). Since their calculation depends on a certain compiler structure (the loop tree), they can be computed only after certain phases in TPO. This situation is common for the majority of modern compilers/optimizers, and a number of transformations may work without any loop tree information. At the same time, more engineering efforts were needed to implement these features. That is why their calculation was made separately from the first group of features; thus, we could use either any single group or the whole set. A somewhat similar set of features where used for the unroll factor prediction [16] by *Stephenson and Amarasinghe*.

In Table 3.1 we show all 29 features that we used in this research. We do not state that it is the best possible set of features, but the features themselves and their number compared to other works [16, 6, 1, 17] appears sufficient to describe a method. Using these features for every method we define a feature vector $\bar{\mathbf{x}}$ that represents this method for our classifier. For example, the *resid* method inside the *mcf* SPEC2000 [8] benchmark has the feature vector $\bar{\mathbf{x}} = \{1379, 558, 0.4, 20, 0.01, 64, 0.05, 10, 0.01, 16, 0.01, 0.01, 222, 0.16, 0.05, 2, 6, 6, 1, 0, 1, 1, 0, 0.33, 1, 0, 1, 33, 227\}$.

Feature	Explanation
Total	The total number of instructions
Loads #,%	The number/percentage of load operations
Store #,%	The number/percentage of store operations
Float #,%	The number/percentage of floating point operations
Cmps #,%	The number/percentage of comparison operations
Branches #,%	The number/percentage of branch operations
Convs %	The percentage of type conversion operations
Indirect #,%	The number/percentage of indirect memory access operations
Long %	The percentage of long-size operations

(a) General features

Feature	Explanation
MaxNestLevel	The maximum nest level in the method starting with 0.
LOOPS #	The total number of loops
Perf.Loops #,%	The number/percentage of the perfect loops
ExacIter%	The percentage of loops with the known number of iterations
Norm %	The percentage of loops that are normalized
WellBehaved %	The percentage of loops that are "well-behaved"
Indep. %	The percentage of loops that are independent
Inner %	The percentage of the inner-loops
Avg. Stmt Count	The average statement number per loop
Avg. Jump Count	The average jump number per loop
Multi-Dim	Multi-Dimensional Access in the loops (binary)
Subscr. #	The total number of subscripts inside the loops
Refs. #	The total number of references inside the loops

(b) Loop-based features

Table 3.1: Set of features used for transformations' prediction(29)

This means that this method has 1379 operations in the TPO Intermediate Representation (IR); it has 64 float operations (that is 5% of the total number), a 2 level nested loop, and multi-dimensional accesses inside the loops. Hence, this is a floating point SPEC benchmark with hot loops that works with the arrays and/or complex data structures. This vector representation can be easily used by different ML techniques, since it is represented in the most traditional way [3]. Clearly, some of the features could be redundant, but this problem should be resolved by our logistic regression classifier by making the corresponding weights relatively small. However, at the beginning of the learning process they can be a source of noise, and, hence, the irrelevant ones should be removed when they prove to be unneeded.

All the features listed are static, so they do not depend on the program input and do not always characterize the method behavior well. Dynamic characteristics like performance counters [5] or even the same features for the dynamic compilers could be even more powerful.

3.3 Heuristic Context

The current method of saving default heuristic values for most compilers is based on the idea of having them embedded somewhere inside the transformation code. However, this has a serious disadvantage - it is hard to control multiple heuristic properties from a central place. Adding new functionality to the compiler can be quite cumbersome and error-prone, because of the need to find and update every heuristic property that is affected by the change.

An alternative way is to record all of the heuristic values in a central data structure, which can be read, updated and dumped to a file quickly. It should have a way to modify the data from the command line, e.g. to override the default values. In addition we need to have a fine-grain control over heuristic values, so that different values can be chosen for individual functions, with no need to modify the compiler source.

The TPO compiler provides all these characteristics using the Heuristic Context and Heuristic Context Modifiers [18]. For each optimization transformation we assign an abbreviation – a unique text string with no spaces that will represent the transformation (e.g. Index-Set splitting transformation has the abbreviation ”ixsplit” and Unrolling transformation has ”unroll”). For each transformation the set of heuristic properties was defined, controlling the ”behavior” of the transformation, and an abbreviation of each property was assigned.

Each instance of a heuristic property has the following components:

abbreviation - a short name that is used to represent the property (”abbrev”);

transformation abbreviation - a short name for current transformation in which this instance is defined (”transabbrev”);

default value - the default value for the heuristic property;

range - the range of legal values for this heuristic property. For example, {0,1} for Boolean type property, or {1-6, 10-20} for integer type property (with possible sub-ranges);

description - a short textual description for the heuristic property.

Heuristics Property instances can be accessed by a key composed of ”transabbrev”.”abbrev” (e.g. ”unroll.enabled” allows to enable/disable unrolling transformation). To modify the Heuristic Context and its properties we can use an Heuristic Context Modifier, which is a set of triplets (key, operator, value). Operator is one of {”=”, ” $<$ ”, ” $>$ ”}, and key, and value are just strings without spaces. For our project we used only ”=” operator. For example, if we want to consider register pressure while unrolling loops we just need to use ”unroll.regpr=1”. Here the ”unroll” string is the transformation abbreviation; the ”regpr” string is the property abbreviation for the register pressure factor; and we use "=" operator to set the property value to "1", which enables this heuristic property. More details about the Heuristic Context approach can be found in *Method and system for managing heuristic properties* [18].

3.4 Choosing the Transformation Set

After obtaining a method description with the features, we need to choose which transformations to optimize. We are interested in those that are (i) included at the *-qhot -O3* level, (ii) not included at the *-O3* level, (iii) take a significant amount of time to compile, and (iv) influence the performance of the generated code. With the help of the TPO compiler writers we formed Table 3.2, which represents the transformations in which we are interested. This set can be divided into three categories: heuristics that either influence the usage of all other transformations (Loopnestonly) or operate with a big set of transformations (Loops and LSCALS), late scalar optimizations and loop optimizations. Every transformation from the last two groups has at least one heuristic property called *.enabled* which is binary. If it is set to 0, then this transformation will be disabled, and will be enabled otherwise. For example, the *SCALS.MAXPASS* heuristic property, which defines how many times we will apply the set of scalar optimizations, usually takes values from 2 (for *-O3* level) up to 5 (for *-qhot -O3* level).

There are a total of 24 heuristic values that we are trying to predict. This gives at least 2^{24} different variants if we assume that every property is binary. Of course, some properties are not binary, and this gives only the variants for a single method. For the case where we have N hot methods for a test we will get 2^{24*N} as a lower bound on the number of variants. Clearly it is not possible to try every variant. Hence, one of the main problems is how to find the best values in such a huge search space.

Transformation	Explanation
Loops	All loop transformations
Loopnestonly	Apply all transformations to loop nests only
LSCALS	Late Scalar Optimizer Set
SCALS.MAXPASS	Maximum number of passes for the scalar transformations set

(a) Complex Heuristics

Transformation	Explanation
LOOPVRP	Loop Vertical Routing
ICM	Invariant Code Motion
UNSWS	Unswitching
CSEFF	Common Subexpression Elimination For Float

(b) Late Scalar Transformations

Transformation	Explanation
ITCNTARR	Array Iteration Counters
INDEP	Mark Independent
UNROLL	Inner Loop Unrolling
DUMLOAD	Dummy Load Insertion
PARTREDUC	Partial Sum Reduction
BALTREES	Tree Balancing
IDIOM	Init/copy Idioms
REDUC4	Reductions
CUNR	Complete Unrolling
VECTS	Vectorizer Set
NVSPLIT	Node Splitting for Vectorization
BGATHER	Gather for Blocks
IXSPLIT	Index Set Splitting
VERSION	Versioning
WANDWAVING	Wand Waving
BLOCK	Loop Blocking

(c) Loop Transformations

Table 3.2: Descriptions of the 24 transformations and heuristics that were used.

3.5 Gathering Training Data

As we mentioned in the previous section the problem of finding the best set of transformations for the current test can be a serious problem, because of the search space size. But we still need to solve it in some way if we want to start the learning process. Other works [6, 16] proposed either a full search for small spaces or the use of randomly generated heuristic values. The full search is not suitable for our goals, because of the search space size. The random set can be too unpredictable as to the quality of the result. Several experiments to gather training data with the random approach did not give any significant decrease in the compile time. However, the peculiarity of the problem that we are trying to solve allows us to use another approach for searching.

The TPO optimizer has a fixed order of transformations such that the next transformation uses the output from the previous one as its input. This property suggests two approaches: (1) start at the *-O3* level and enable new transformations, (2) start at the *-qhot -O3* level and disable transformations . We decided not to enable new transformations starting at *-O3* level. In this case we need to find the exact set of transformations that decreases the execution time to the acceptable time of the *-qhot -O3* level, and this can be hard with the random approach. Instead, we can disable transformations starting with the full set at *-qhot -O3* level and control that the execution time does not increase significantly and the compilation time gradually decreases too. If we start disabling transformations backwards (from the last till the first) we are less likely to break a useful queue of transformations that gives a speedup.

We do not state that this always gives the perfect values for heuristics. For example, suppose one early transformation X does something bad with the IR (as to the performance), and then transformation Y makes it okay again later. In our approach we first will try to disable Y, and this will be unsuccessful, because Y does something good as to our assumption; then we will disable X (assuming that it does not do something good). At the end we will have transformation Y enabled, but it is unnecessary after X is disabled. However, this approach allows us to find the best (or nearly the best) set of transformations that linearly depends on the number of heuristic parameters, which makes it possible to be used in practice.

Now we are going to describe our algorithm in pseudo-code. The formal description of this approach with the appropriate comments is in Figure 3.1.

```
GenerateTrainingData($tests, hots, trans$)
 1  for each test in tests
 2      do
 3         Init($curr\_settings[test], best\_settings[test]$)
 4         for $i \leftarrow length[trans]$ downto 1
 5             do
 6                for each method in $hots[test]$
 7                    do
                         ▷ Disable current transformation for method in test
 8                       $curr\_settings[test][method][i].enabled = 0$
                         ▷ Run test and get the compile and execution time
 9                       $result = Run(test, curr\_settings, \&curr\_comp, \&curr\_exec)$
10                       if IsBetter($result, curr\_comp, curr\_exec$)
11                          then Update($best\_settings[test], curr\_comp, curr\_exec$)
12                          else $curr\_settings[test][method][i].enabled = 1$
```

Figure 3.1: The algorithm for Generating Training Data

For every test we have a set of hot functions that we obtained by using the *tprof* tool. Using the Heuristic Context Modifiers mechanism we can disable/enable certain transformations. After the execution of the current test (line 9) we have results: the test correctness, the compile and the execution time. Next, we check whether the current execution was better than previous ones. This is done by the *IsBetter* function (line 10) call.

The *IsBetter* function checks whether the new execution time is not significantly (e.g. 1%) worse than the best execution time and the baseline execution time. The same checks are performed for the compile time. The baseline check is needed to avoid a gradual degradation. For example, a 1% increase in every execution during 200 runs can cause a 3 times increase in the execution time. The compile time should normally decrease all the time, but there can be two cases when this does not happen. The first one is a simple time fluctuation that is the result of limited accuracy of the time measurement. It may happen even with the same executable when run twice. The second one is that disabling of one transformation can cause others to perform much more work and, hence, increase the compile time.

3.6 Learning Process

After gathering the training data for every method we have a feature vector $\bar{\mathbf{x}}$ and the corresponding vector of the best transformation set $\bar{\mathbf{C}}$. Our goal is now to find the function:

$$F(\bar{\mathbf{x}}) = \bar{\mathbf{C}}$$

or at least to approximate it in some way. With this function $F(\bar{\mathbf{x}})$ we then, having a new feature vector $\bar{\mathbf{x}}'$, can approximate its best transformation set $\bar{\mathbf{C}}'$ with some good $\bar{\mathbf{C}}''$.

3.6.1 Classification Algorithms

Our problem is the classical ML problem and several powerful methods were invented to solve it. One of the simplest is the nearest neighbours (NearN) or K-nearest neighbours. To classify a test feature in this method you need to calculate the most common class amongst its K nearest neighbours in the training set. One of the questions here is how to measure the distance? The usual way is to use the squared Euclidian distance $d(m,n) = \sum_d (x_d^m - x_d^n)^2$.

The second popular classification method is based on artificial neural networks (ANN) or simply neural networks (NN) [3]. Historically it was motivated by relations to biology, but for classification purposes NN are just nonlinear classification machines.

In the simplest case with the linear activation function, the NN reduces to a linear network equivalent to a logistic regression. The gradient of conditional likelihood can be easily computed for NN with 1 hidden level, using the efficient backpropagation algorithm [14]. The main disadvantage of this approach is that we loose convexity property, and, hence, can find a local minima instead of a global one.

As we mentioned above in Section 2.2, in our research we used logistic regression with penalty regularization as an easy and an effective way for classification. We have a 29 dimensional space of features and a 24 dimensional space of outputs that can be considered as 24 single outputs for simplicity; and we do not have any prior knowledge (like sparsity or features' dependencies) about our training data to apply something special. The next section describes logistic regression in more detail.

3.6.2 Logistic Regression

Logistic Regression is a popular linear classification method. Its predictor function consists of a transformed linear combination of explanatory variables.

The logistic regression model consists of a multinomial random variable y, a feature vector $\bar{\mathbf{x}}$, and a weight vector θ. In our case y is the possible value of the heuristic property. We initially set the weight vector θ with some random values, later, they will be changed with the ones that maximize the *conditional log-likelihood* (3.2). y's posterior is the "softmax" of linear functions of the feature vector $\bar{\mathbf{x}}$, as shown in Equation 3.1.

$$p(y = k|\bar{\mathbf{x}}, \theta) = \frac{\exp(\theta_k^\top \bar{\mathbf{x}})}{\sum_j \exp(\theta_j^\top \bar{\mathbf{x}})} \tag{3.1}$$

To fit this model we need to optimize the *conditional log-likelihood* $\ell(\theta; D)$:

$$\ell(\theta; D) = \sum_n \log p(y = y^n|\bar{\mathbf{x}}^n, \theta) = \sum_{nk} y_k^n \log p_k^n \tag{3.2}$$

where $y_k^n \equiv [y^n == k], p_k^n \equiv p(y = k|\bar{\mathbf{x}}^n)$.

To maximize the log-likelihood we set it's derivatives with respect to θ to zero.

$$\frac{\partial \ell(\theta)}{\partial \theta_i} = \sum_n (y_i^n - p_i^n)\bar{\mathbf{x}}^n \tag{3.3}$$

Typically some method like conjugate gradients [15] can then be used to maximize log-likelihood. It needs (3.2) and (3.3): value of ℓ and it's derivatives.

For the case when we have one binary output at a time, these formulas (3.1, 3.2, 3.3) can be simplified:

$$p(y=1|\bar{\mathbf{x}},\theta) = \frac{1}{1+\exp((\theta_0^\top - \theta_1^\top)\bar{\mathbf{x}})} \tag{3.4}$$

$$\ell(\theta; D) = \sum_n \log p(y=y^n|\bar{\mathbf{x}}^n,\theta) = \sum_n y^n \log p^n + (1-y^n)\log(1-p^n) \tag{3.5}$$

where $y^n \equiv [y^n == 1], p^n \equiv p(y=1|\bar{\mathbf{x}}^n)$.

Regularization

For supervised learning algorithms, over-fitting is a potential problem which we need to solve. It is a well-known fact that for unregularized discriminative models fit via training error minimization, sample complexity (training set size needed to learn "well" in some sense) grows linearly with the Vapnik Chervonenkis Dimension (VCD) [19]. Moreover, the VCD for most models grows linearly in the number of parameters, which usually grows at least linearly in the number of input features. So, if we do not have training set size large relative to the input dimension, we need some special mechanism, such as regularization, which makes the fitted parameters smaller to prevent over-fitting [13].

The most two well-known regularization methods are *Ridge Regression*, which uses L_2 penalty function and *Lasso*, which uses L_1 penalty function. L_2 penalty function uses the sum of the squares of the parameters and Ridge Regression encourages this sum to be small. L_1 penalty function uses the sum of the absolute values of the parameters and Lasso encourages this sum to be small. We can penalize the logistic regression from 3.2

using the linear combination of a special function ψ for every parameter θ_k in a such way:

$$\ell^*(\theta) = \ell(\theta) - \lambda J(\theta) \tag{3.6}$$

where

$$J(\theta) = \sum_k \alpha_k \psi(\theta_k), \tag{3.7}$$

$\alpha_k > 0$.

Usually this function ψ is chosen to be symmetric and increasing on $[0, +\infty]$. Furthermore, ψ can be convex or non-convex, smooth or non-smooth. A good choice for this function should result in *unbiasedness*, *sparsity* and *stability*. We tried both L_1: $J(\theta) = \sum_k |\theta_k|$ (or ℓ_1) and L_2: $J(\theta) = \sum_k \theta_k^2$ (or ℓ_2) in our experiments. However, we did not notice any significant changes depending on the penalty function type, and, hence, we finally chose one of them (L_2) to present our results in Chapter 5 .

Cross Validation

In all our experiments we used a traditional cross-validation technique. That is, we computed an optimum vector θ on one set of tests and then used it to measure the results for other tests. Hence, no train and test data intersection happened and thus the results should be fair.

3.7 Deployment Process

After finishing with the learning process, we have a vector of optimum parameters θ that we can use to predict the transformation set using the feature vector $\bar{\mathbf{x}}$. To achieve this we just need to use Equation 3.4 for every transformation. If the probability is bigger than 0.5, then we will apply this transformation, and we will not apply it otherwise. Sometimes researches start to apply a transformation with a larger threshold (e.g. $p > 0.6$ [6]), but this needs further investigation and it is not clear to what value the threshold should be set.

In this chapter we described the basis of our work - all the approaches and ideas that were used to reach the goal. We explained how a method could be represented using features; how the compiler decisions could be ruled externally using the Heuristic Context approach; which transformations were an object for prediction; and how we gathered training data, and, then used it in the learning and deployment processes. We also provide enough details as to the ML techniques used to make our experiments easy to reproduce.

The next chapter will provide more technical details about the implementation, infrastructure and methodology we used for executing the set of experiments. However, all the experimental results with the appropriate discussion will be presented in Chapter 5.

Chapter 4

Infrastructure

God is in the details.

— Gustave Flaubert

This chapter describes the implementation details for our approach (framework overview, compiler changes and logistic regression implementation) and the setup for all our experiments. However, the experimental results will be provided in the next chapter.

4.1 Framework Overview

Every approach needs an implementation to be evaluated. In our case we need a framework that consists of several parts: changes inside TPO, which we call the *compiler part* of the framework; the part where all the training information can be collected, which we call the *middle layer*; and the logistic classifier part, *ML part*.

The first part should be implemented in the same language as TPO was implemented (C++), because it consists of changes inside of the optimizer. For the other two parts we had a certain freedom in implementation. The middle layer needs to perform the benchmark executions, measure the compile and execution time, and then change the options appropriately. Hence, most of the scripting languages, e.g. *perl*, are a suitable choice for the implementation of this part. The ML part can have a variety of different choices for implementation, but we chose Matlab where it is relatively easy and efficient to program matrix-based calculations.

4.1.1 Compiler Part

As we mentioned above, we need to make several changes inside TPO. The main problem is the absence of the *function-level granularity* that we need to change Heuristic Context (the values of the heuristics) for every single function. The second problem is how to incorporate the feature gathering inside the existing optimizer. Now we describe how these problems were solved.

Function-Level Granularity

A compiler user may need to change heuristic values for functions differently, and our project also needs this possibility. To provide this flexibility we added new options inside TPO that provide a notion of a *mode*. We can then specify all heuristics for the current mode, all functions for the current mode, and the default mode for functions that do not have any special mode.

We defined several new options
hmodifier option has the syntax:

$$hmodifier = mod_1, mod_2, ..., mod_N : mod_1 = f1, f2, f3 : mod_2 = f10, ..., mod_N = f7, f4$$

hmodpath and *hpredict* options have the the same syntax:

$$hmodpath = /home/.../hmod\_dir/$$

The first option specifies a set of modes and a set of functions for every mode. For every mode we should have a file in the *hmodpath* directory called *mode_name.hmod* and one additional file for all functions that do not have any special mode called *default.hmod*.

```
Heuritic Property name:
# Options for func mode
CSEFF.ENABLED=0
ITCNTARR.ENABLED=0
UNROLL.ENABLED=0
DUMLOAD.ENABLED=0
BALTREES.ENABLED=0
VECTS.ENABLED=0
BGATHER.ENABLED=0
IXSPLIT.ENABLED=0
VERSION.ENABLED=0
WANDWAVING.ENABLED=0
SCALS.MAXPASS=2
```

Figure 4.1: Hmod file example with heuristic properties

Every file should have a set of heuristic values properties set to some values according to the Heuristic Context rules as shown in Figure 4.1 ("#" is used for comments). The *hpredict* directory should have a file for every heuristic we want to use in TPO e.g. *loops.txt*, *unroll.txt*. Every file should have a set of parameters learned with the logistic regression classifier. The number of features that were used in the training phase, and, then, learning and usage, should be the same.

The implementation of these options needs only a simple string parser that we added inside TPO. The only possible problem with this approach is that function names can be long, especially for C++, and every compiler has a limitation on the input string length. This can be solved by replacing a function name with a short unique abbreviation; and all the abbreviations should be in the special file of the format like *filename1 fn_abbrev1* for every function that we are going to use with modes.

FeaturesCollection Class

In order to collect the information about method features, we created a special C++ class called *FeaturesCollection*. It was made as to the TPO rules for a single transformation and can be called as a regular optimization in the transformation list. Every feature has a member in the *FeaturesCollection* class (e.g. *int32 opersNmbr* for the total number of operations or *double pBranches* for the percentage of branches).

The feature computation is separated into two functions called *GatherFeatures* and *CollectInnerLoopInfo*. The first one is used to calculate the relatively simple features that do not need the loop tree to be computed and can be called anywhere in the code when the current procedure (*Procedure *pProcedure*) is valid. The second one is used to calculate features that are "loop tree" dependant, e.g. the maximum nest level or the presence of the multi-dimensional array access in the loops. *CollectInnerLoopInfo* needs a current procedure, *Procedure* class, and a loop structure, *LoopOptimizer* class, as input parameters.

The *FeaturesCollection* class also provides functionality to calculate a single transformation prediction that can be computed using the parameters from the logistic regression. The *CountPrediction* method gives an answer to the question whether a transformation should be applied to the current method or not. Using this function for every interesting transformation, we generate the Heuristic Context that will be used for the current method.

To add a new feature you need to add a class member in the header file and write code to calculate it in the corresponding *cpp* file. The feature gathering has a switch option called *features* that is switched off by default to avoid unnecessary overhead. The whole class implementation is about 1000 lines of C++ code that was written separately from the rest of the TPO code.

4.1.2 ML Part

The ML technique called logistic regression was implemented in Matlab [11] and was tested with several data sets from the UCI Machine Learning Repository [2]. Most of the details were provided in Section 3.6.2 in the previous chapter.

Matlab Implementation

Our current classifier was implemented in Matlab and consists of the several *.m* files that allow us to calculate likelihood and it's gradient (*likelihood.m*), extract data for different data sets (*data_"testname".m*), and perform minimization (*minimize.m*). The main method called *lgc_f* allows us to change the number of iterations in minimization, choose the type of regularization (1 - for L1 and 2 for L2 see Chapter 3), and the parameter λ as the coefficient for regularization.

For minimization we used the Polack-Ribiere flavour of conjugate gradients [15], and a line search using quadratic and cubic polynomial approximations. The Wolfe-Powell stopping criteria is used together with the slope ratio method for guessing initial step sizes. The Matlab implementation of minimization was made by *Carl Edward Rasmussen*.

4.1.3 Middle Layer

The changes inside the compiler do not give us the training data that we need to use the logistic regression classifier. In order to unite the compiler and the ML part and gather training data we implemented the *middle layer*. It consists of a set of perl scripts that are used to execute benchmarks, evaluate the results and represent them in such a way that they can be used in Matlab for classification.

The first group of scripts is the "baseline" ones that are used to measure the compile and execution time for *-O3* and *-qhot -O3* levels for different benchmarks. They generate results for every single benchmark that will be used by other scripts as the baseline. With the option *features* they are also used to collect method descriptions. The second group of scripts is the ”gather” ones that search for the optimal value for the set of heuristic values. The main algorithm was described in Section 3.5, *Gathering Training Data.* The last group is the ”parse” one. It is used to transform results from the previous group execution into the form that can be used in the classifier.

The scripts are easy to read and modify and could be transformed to work with other benchmarks, not only SPECs.

4.2 Benchmarks

In our experiments we used benchmarks from SPEC2000 [8], SPEC2006 and some additional float benchmarks from IBM customers. For training purposes we used twenty-one benchmarks from SPEC2000. This gave us 140 hot (from the top of the *tprof* tool output) functions that were used to collect features and search for the optimal set of transformations. It is possible to use benchmarks that are written in C, C++, Fortran, and Fortran90.

4.3 Measurements

4.3.1 Platform

Our target platform was PowerPC. For evaluating the results we used one of the IBM servers that has 4 x Power5 1900 MHz Processors, 32 GB Memory, running OS AIX 5.3. We used -02 optimizations in all experiments to compile TPO.

4.3.2 Compilation and Execution Time

Even on a server without any workload the same application execution can take different amounts of time. The variation is usually relatively small, and may depend on many factors. Since we measure and compare the execution and compile time, we need to keep this possible time variation in mind. In order to solve the problem, we consider the time interval, not just fixed time moment. If we want to compare two "times" we compare time intervals instead. If intervals intersect then they can be considered as the same time for our purposes. The interval time is generated as a small percentage from the base one, e.g. for the execution time 100 seconds we will have the time interval 99 - 101 seconds (1% possible fluctuation).

The size of intervals was found approximately during our executions. Currently we use 1 % for the possible fluctuation for compile time and 0.5 % for the execution time. Such a comparison may lead us to miss a small performance degradation. This is not a great problem if it happens once per benchmark, but we need to avoid the accumulation of a number of such degradations. To achieve this we compare every new time with the previous best and the baseline, so it is impossible to become significantly worse than the baseline.

In this chapter we described the infrastructure of our project, including all valuable technical details of implementation for both the compiler part and ML part of the project. We introduced the special middle layer that was needed to gather training data and to perform the interaction between layers. We also provide information about the benchmarks used for experiments, and some measurement-specific details.

In the next chapter we will provide our results as to the training data gathering that shows possibilities for predicting with the current search approach. Then we evaluate our predictor quality for both the compile and execution time, and discuss why we obtain certain results and how they can be improved.

Chapter 5

Results and Discussion

People love chopping wood. In this activity one immediately sees results.

— Albert Einstein

In this chapter we present the results we achieve during our experiments. The first part consists of the training data results, which are not only necessary for the learning algorithm, but can also be used to analyze possible improvements and final speedup. Then, we present the current prediction quality by showing how well we perform on different tests: SPEC2000 that we used for training, other SPEC2000 tests, and non-SPEC tests from the IBM customers.

5.1 Training the Classifier

As we mentioned above, we use SPEC2000 for most of our experiments. For training purposes we chose 21 tests out of the 26 SPEC benchmarks.

The first question that arises when we have our optimal or sub-optimal heuristic values is how well can we decrease the compile time. It is unrealistic to expect the predictor to show better results than the optimum we found, hence, if the training results are not good enough it does not make sense to evaluate our classifier.

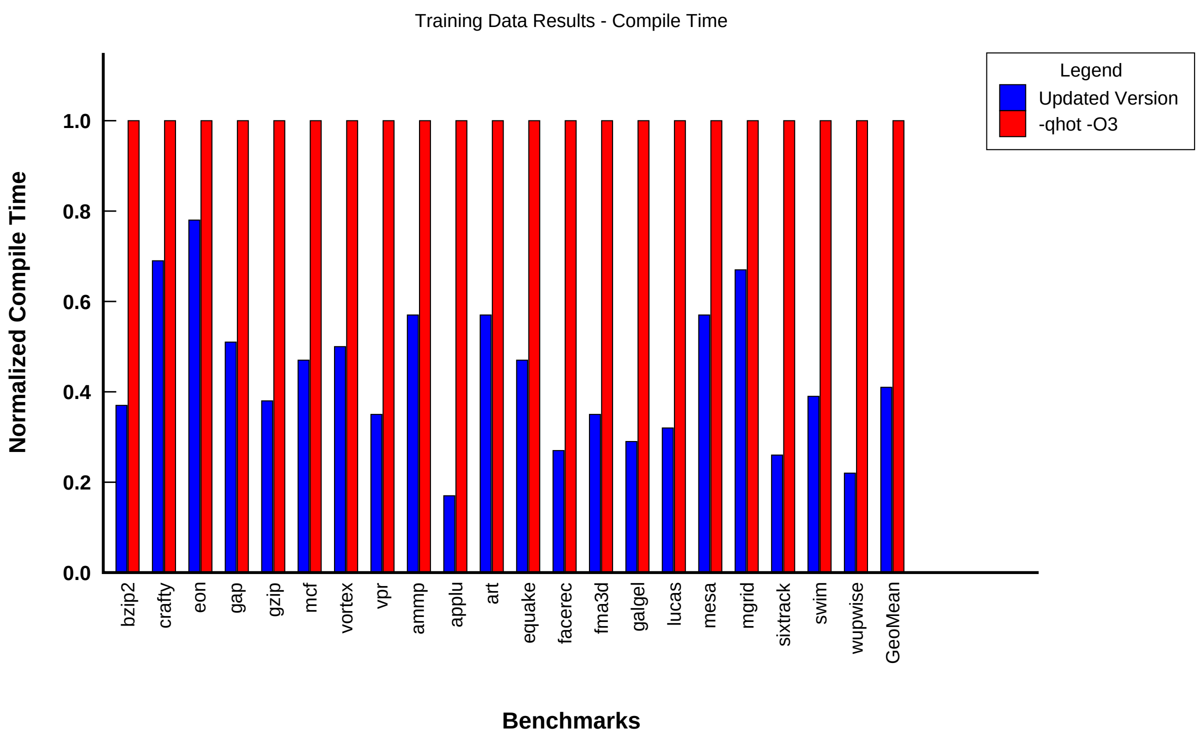

Figure 5.1: Comparison of the compile time for the new version with the set of heuristics after training vs the original version at -qhot -O3 optimization level

Figure 5.1 shows the compile time for the set of optimal heuristic values, normalized to the compile time of the *-qhot -O3* level, to highlight the reductions achieved. Table 5.1 provides the raw compile times for both approaches. Of course, we have different results for every test, but a significant speedup can be achieved for every benchmark starting with a 1.30 speedup for the *eon* benchmark up to a 6 times speedup for *applu*. A geometrical average speedup for all tests is **2.46**. If we consider the SPECfp only, then we achieve a *2.75* times decrease in the compile time. These results look promising, but only provided that we do not significantly slow down the benchmarks' execution.

Figure 5.2 proves that we can efficiently find the optimum heuristic values such that we do not have any significant increase for the execution time. Moreover, we can even decrease the execution time for certain tests: gap, applu, mesa, sixtrack, swim and wupwise. The overall result is a *1.02* speedup, which is comparable with the results

Test Name	Updated Version, sec	Baseline, sec
bzip2	3.88	10.55
crafty	31.76	46.03
eon	98.67	127.22
gap	90.12	177.79
gzip	5.14	13.37
mcf	2.1	4.47
vortex	42.4	85.02
vpr	12.97	37
ammp	27.51	48.15
applu	10.1	60.7
art	3.05	5.31
equake	3.66	7.78
fma3d	100.76	284.34
galgel	22.5	77.57
lucas	12.63	39.79
mesa	77.61	135.49
mgrid	6.44	9.55
sixtrack	128.27	496.38
swim	1.13	2.92
wupwise	3.36	15.3

Table 5.1: Comparison of the compile time for the new version with the set of heuristics after training vs the baseline

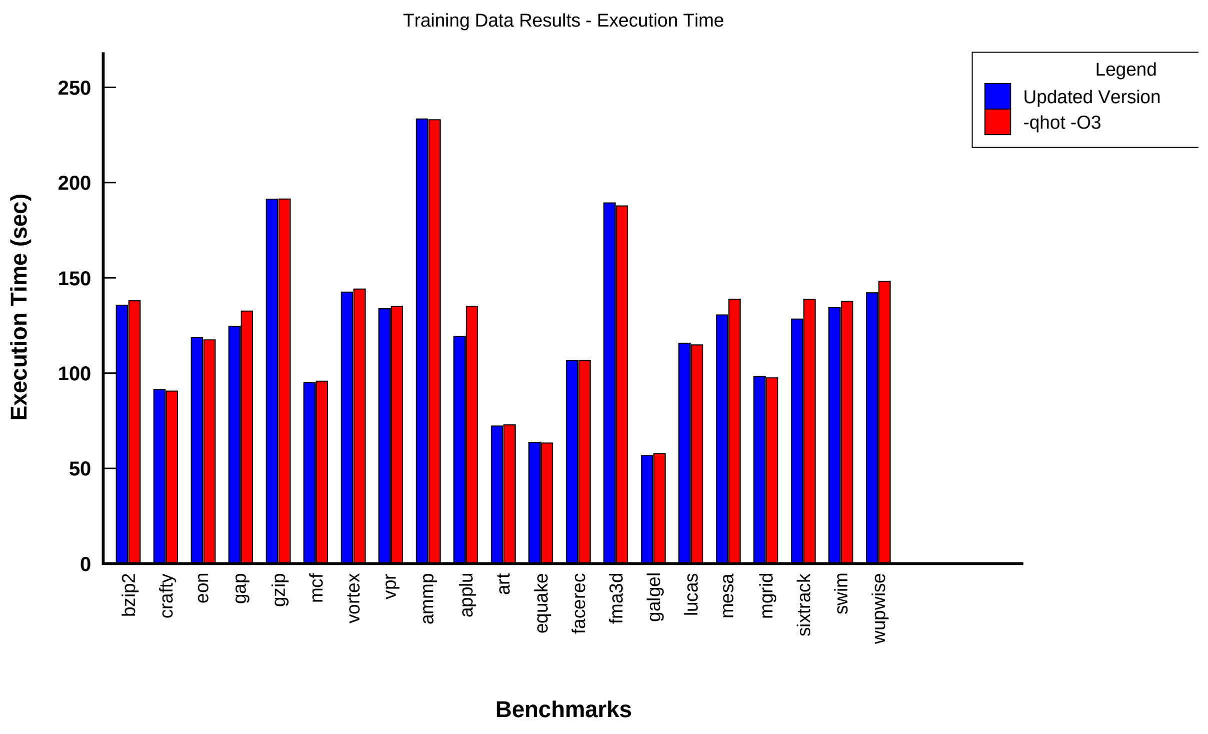

Figure 5.2: Comparison of the execution time for the new version with the set of heuristics after training vs the original version at -qhot -O3 optimization level

from several other works ([16, 17]) where execution time speedup was the primary goal. We should keep in mind that these results were obtained using a well-tuned commercial compiler, TPO, in which significant effort has been made to show a good performance level on the SPEC benchmarks.

Another interesting question is how often a transformation was needed among all the benchmarks. The answer is that it depends on the benchmark type. The SPEC CPU benchmarks are divided into two categories: the integer benchmarks (SPECint), and the floating point benchmarks (SPECfp). For the SPECint set of benchmarks the transformations were unnecessary in most cases except for the complex ones, e.g. LOOPS, LSCALS. The situation is reversed for the SPECfp benchmarks where usually at least several different transformations were needed to obtain good execution time. We also found that the *eon* benchmark, which is considered as part of SPECint [8], has a significant number of the float operations in its hot functions (from 10% to 25%). Hence, it is more convenient to consider this benchmark as a float one.

This data can be used for function classification and clustering even without any machine learning algorithm. We can define a set of transformations that are needed for a certain type of function, and, thus, decrease the compile time. However, it is not easy to do so in the general case and it is less powerful and flexible than machine learning approaches.

One of the most serious drawbacks of the results we obtained is that each single transformation is needed in only a very few cases, usually for less than 5% percent of the hot functions. It makes these rare cases, when a transformation is necessary, look like noise, and reduces both the stability and the accuracy of our classifier. At the same time we have an unavoidable noise, because we measure the real time and fluctuations may easily occur. With such a usage level it is hard to expect any predictor to be very efficient. In order to make the classifier perform well in a general case we need a more serious set of training data, not only SPEC benchmarks.

The results of these runs were used to train the logistic regression classifier as described in Section 3.6.2. The results of the classifier were incorporated into TPO as described in Section 3.7 and Section 4.1.

5.2 Classifier Evaluation

Our main goal was to predict whether a transformation was needed for a given function or not. The training data gives us this answer, but it can be considered only as an *Oracle*, because we can not afford so many executions to collect this information. Hence, we need to apply our logistic classifier and evaluate its prediction quality. There are two possible ways to perform this evaluation: offline and online evaluation.

5.2.1 Offline evaluation

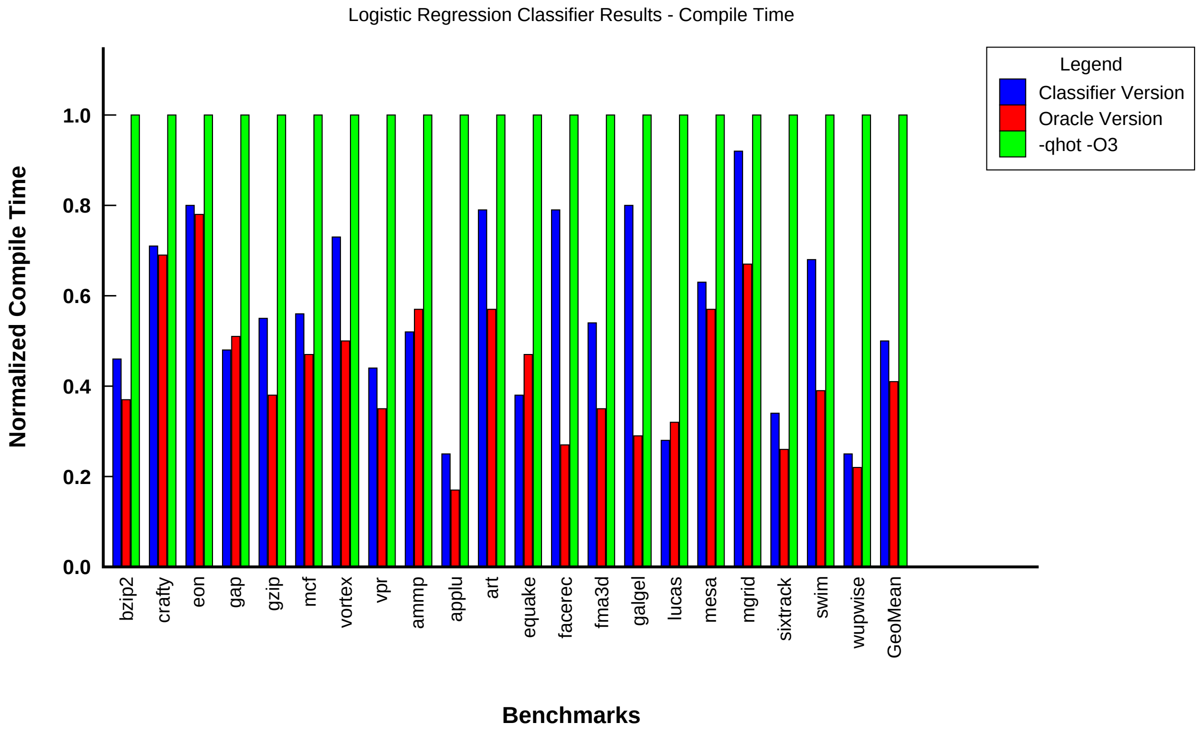

Figure 5.3: Comparison of the compile time using the classifier, the oracle, and the baseline.

Offline evaluation allows us to measure how many errors our classifier will have on the test set. For this set we should know a set of necessary transformations from any source. In practice this means one of two options. The first one is to run our training scripts for another set of benchmarks, which will be time-consuming. The second option is to use the training data we have as the test data excluding every single function from the training set when "testing" it. This technique is called "leave-one-out cross validation" (LOOCV); it allows us to be fair in measuring the results (we exclude a function we test from the training set), and it does not need additional data for testing.

The LOOCV method is easy to implement - we already have all necessary components to do it, but it has one serious drawback - it does not show the qualitative effects of possible mistakes, only quantitative aspect. We can easily see how many errors in prediction we have, but we do not know how exactly they affect the execution time, because this is the offline evaluation.

There are also two type of possible errors. The first one is when we predict that a transformation was needed, but it was not. This leads to an increase of the compile that could be avoided and can be considered as a ”conservatism” in decision making. The second type of error is the opposite case when we say ”not needed”, when it should be ”needed”. This usually leads to a significant increase in the execution time and should be considered a more serious error.

In our experiments we mostly considered the errors of the second type (execution time increase). The error rate depends on the heuristic property type and is higher for the complex heuristics, e.g. LOOPS.ENABLED. For all heuristic properties it was in the 0% - 4% interval. Clearly, the error rate depends on the learning process, which itself depends on the initial value of a vector θ (see Section 3.6.2). This initial value is chosen randomly in certain interval for each component, e.g. [-1,+1], and, hence, the learned parameters may vary together with the error rate based on them. That is why we do not report the full table of error rates for all heuristic properties.

5.2.2 Online Evaluation

The most interesting evaluation is the online one when we have incorporated the classifier inside TPO and measured the results. For this case we used the parameters, which were learned on the whole set of the hot functions, and then use them for each function, not only the hot ones. This means that our training and testing set has a small intersection, which may be unfair. To check whether this can be a problem we tried (for several randomly chosen benchmarks) to exclude their hot functions from learning, and then evaluate these benchmarks using these parameters. The results were the same to within the accuracy of our measurements. In Section 5.3, we show the result of using these parameters for the benchmarks that were not used in the training process.

Figure 5.3 and Figure 5.4 present the results we obtained on the SPEC2000 benchmarks. The compile time average shows a **1.99** times speedup, which is worse than the Oracle (training data), but still allows us to decrease the compile time almost two times. We were less conservative in the prediction for the SPECint, because they used our transformations infrequently.

In some cases we have a compile time that is even smaller than with the Oracle, but this does not mean that this is a positive situation. This means that we disabled something that was valuable, and this usually leads to a significant increase in the execution time. For the *ammp* and *art* benchmarks we obtained a slowdown (18% and 10% respectively) because of the classifier mispredictions. However, in general we had a satisfactory prediction level and achieved only a 1% increase in the execution time compared with the baseline (*-qhot -O3*).

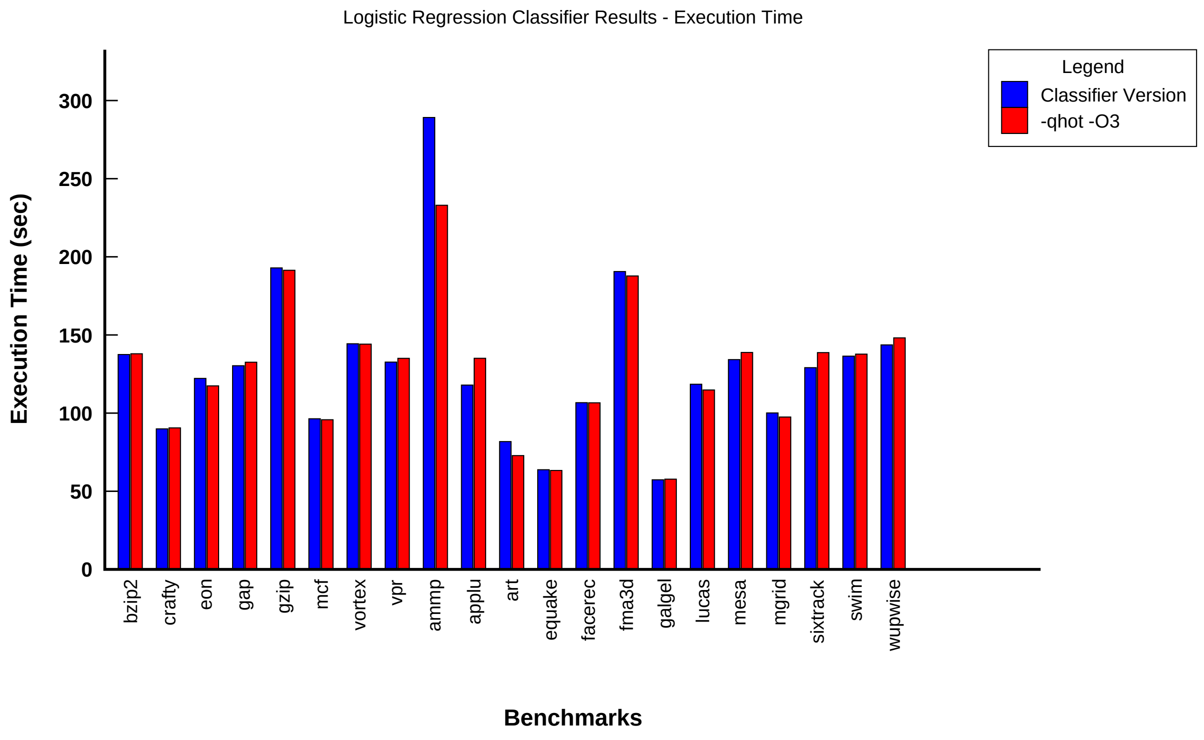

Figure 5.4: Comparison of the execution time using the classifier and the baseline.

Hence, we decrease compile time almost two times with negligible increase in the execution time.

5.3 Other Benchmarks

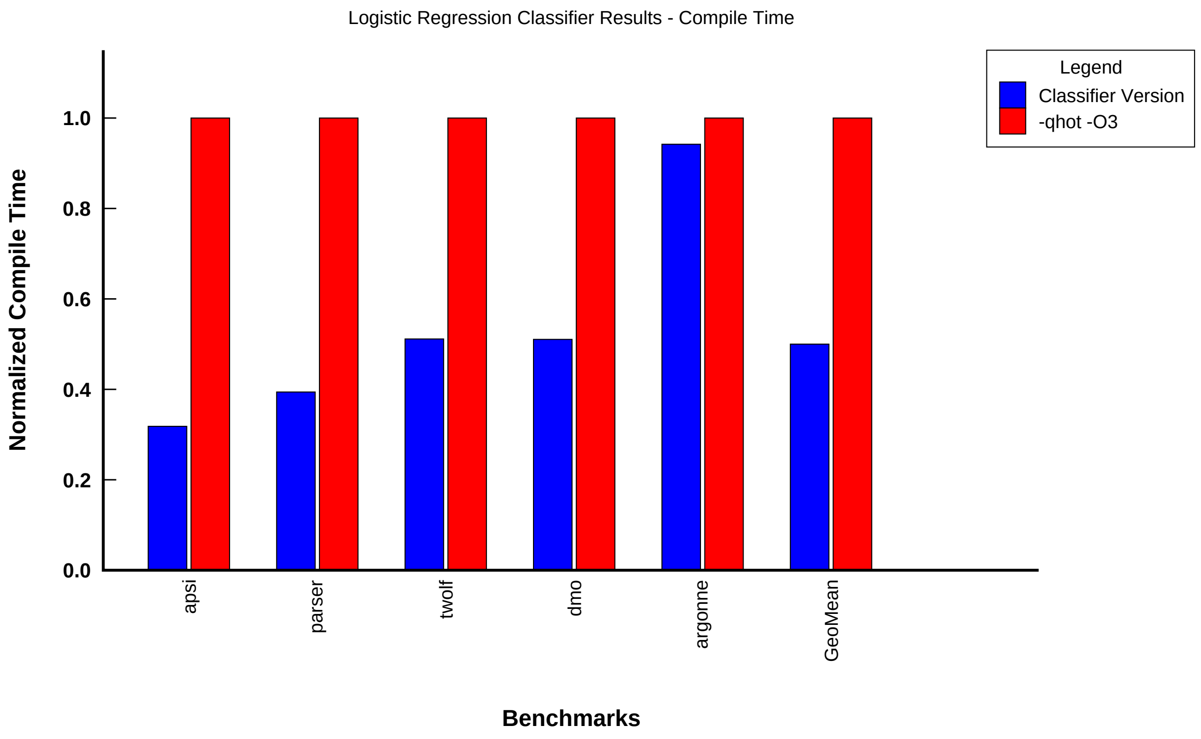

Figure 5.5: Comparison of the compile time using the classifier and the baseline.

In order to show that our classifier works not only for the tests that we chose for training, we used three additional tests from SPEC2000 (*apsi*, *parser*, *twolf*), and two benchmarks from IBM customers called *dmo* and *argonne.* The first three benchmarks were not used in the training set for different reasons. For example, due to a very "flat" profile with an absence of hot functions, or the need for special options to ensure an exact output match.

Our measurements showed very similar results for this set of benchmarks (Figure 5.5, Figure 5.6, and Table 5.2). We achieved a *2* times average speedup for the compile time and *1.04* speedup for the execution time. The average speedup was because of the *dmo*

Test Name	Classifier Version, sec	Baseline, sec
apsi	14.93	46.92
parser	15.34	38.92
twolf	33.92	66.32
dmo	3.14	6.05
argonne	76.09	80.74

Table 5.2: Comparison of the compile time using the classifier and the baseline.

benchmark, showing that one of the transformations or a whole group leads to an increase of the execution time. This is probably a performance bug and should be analyzed by the compiler writers and/or performance specialists. We do not have a significant slow-down on any of these benchmarks, so we can conclude that our technique is effective in decreasing the compile time without losing the ”code quality”.

In this chapter we presented our experimental results. We showed that we can efficiently decrease the compile time using the logistic regression classifier without significant increase in the execution time. The next chapter will provide information about related work for searching both single and multiple heuristics. We also make a comparison with our approach and the results we achieved.

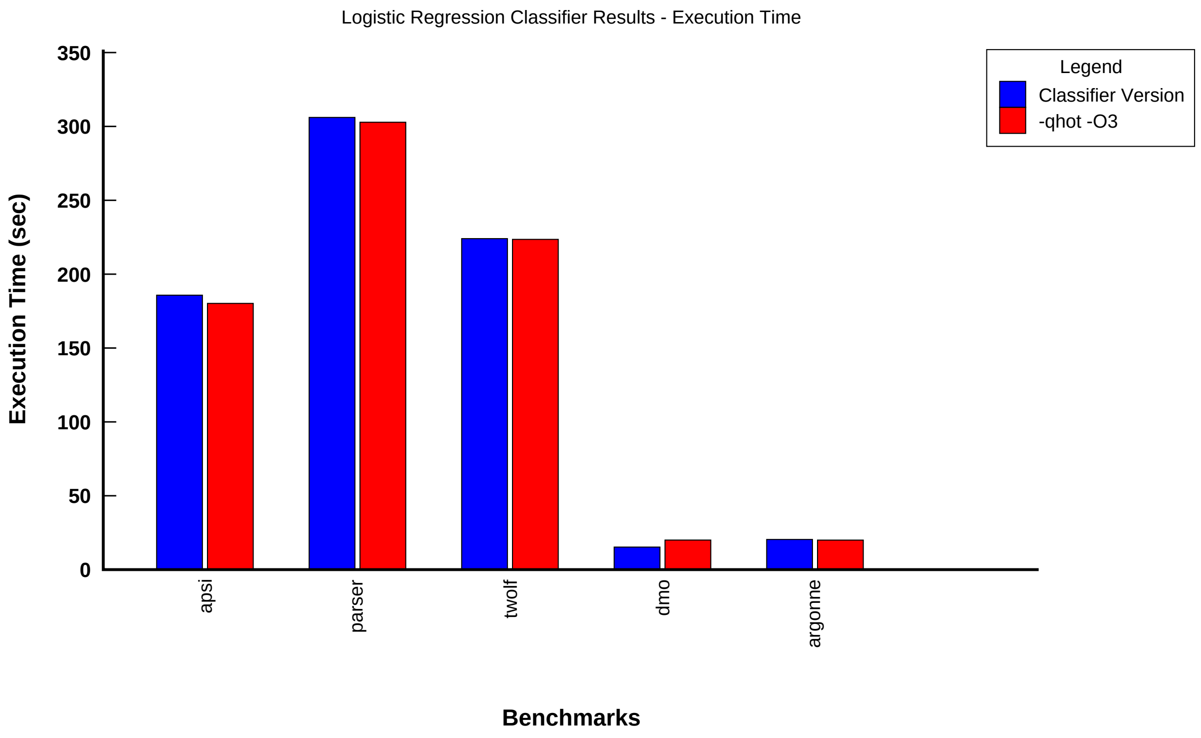

Figure 5.6: Comparison of the execution time using the classifier and the baseline.

Chapter 6

Related Work

If I have seen further it is by standing on the shoulders of giants.

— Isaac Newton

In this chapter we provide an overview of the research papers, which are related with our ideas and/or approach. The chapter is divided into three sections: single heuristic learning, multiple heuristic learning and iterative compilation.

6.1 Single Heuristic Learning

During the last several years there has been a significant amount of work in compiler optimization that tries to apply machine learning to find out the best parameter value for single heuristics.

One of the first works, by Calder *et al.* [4], focused on branch prediction, which is an important element in improving the performance of compiler-generated code. In order to solve this problem the authors used a body of existing programs to predict the branch behavior in a new program and applied machine learning techniques called neural network (NN) and decision trees (DT). While this approach considered the program a whole unit of processing, later work of others [16, 17, 6, 5, 1] used a single method or a function of a program as the smallest unit for description.

The authors called their approach *evidenced-based static prediction* (ESP); the main idea is that the behavior of a corpus of programs can be used to infer the behavior of new ones. This technique has two advantages. First, it generates predictions automatically, and they can be specialized if needed. Second, using only static information, it automatically determines which pieces of information are useful. For this problem the optimal value for predicting a branch was easily obtained by observing the direction for each branch, and, thus, you do not need a special process to retrieve the optimal values for your training data set as in our approach and in several other works ([6, 5]).

Stephenson and Amarasinghe [16] addressed the problem of predicting unrolling factor for different loop nests. The authors chose supervised learning algorithms: nearest neighbor (NearN) and support vector machines (SVM). A subset of loop features was extracted from the unrolling optimization, and eight different unroll factors (1,2, ... , 8) were used as labels during learning. The authors used the Open Research Compiler (ORC) [9] in their experiments, but it was not stable enough, which significantly limited the number of possible unroll factors. The authors decided to perform supervised learning offline, because they wanted to incorporate the learned classifier into the ORC compiler. Accordingly, it was necessary to measure the runtime information of all innermost loops. However, extracting this information is not easy and usually requires loop code instrumentation, which can significantly influence the execution of the program. The authors invested a substantial engineering effort in minimizing this impact.

Experiments with the classification showed an overall speedup on 19 of the 24 SPEC benchmarks, with a 5% average speedup. This is a good result because the classifier, which can be trained in a few hours, outperformed human heuristics that have been tuned for years. However, the experiments were conducted with the software pipelining optimization disabled, while most of the tuning was performed with the software pipelining kept in mind. Another serious drawback of this work is that this technique is not general and it is not clear how it can be applied for every optimization.

Stephenson *et al.* [17] worked on creating a methodology for automatically fine-tuning compiler heuristics. The authors try to solve the problem of finding good heuristics using machine learning techniques for optimizations such as register allocation, hyperblock formation, and data prefetching. The authors chose to employ genetic programming (GP) because it is useful for searching high-dimensional spaces. GP solutions are human readable, so they can be easily converted into arithmetic equations. Moreover, GP is a distributed algorithm, which makes it possible to dedicate a cluster of machines to searching a solution space. Compiler priority functions (e.g. a ”weight” of each instruction in the list scheduler), which prioritize the options available to a compiler algorithm, are the basis for effective heuristics, so these functions are the main goal of the GP evolution process.

The authors achieved significant improvement for both hyperblock formation and data prefetching (Case study 1 and 3 respectively), but these two optimizations were not greatly tuned before. Their Case study 2, which concerns register allocation, is a good example of an optimization that was previously tuned to a great extent. As a result, the authors could not achieve any significant speedup for most of the benchmarks. One of the main problems with the experiments is that the authors used the ORC, so a number of preexisting bugs in this project forced them to remove several benchmarks from the resulting table.

6.2 Multiple Heuristic Learning

Several recent works tried to optimize the whole set of heuristics, not a single one [6, 5], which is very close to our goals in this work.

Cavazos *et al.* [6] developed a new method-specific approach that automatically selects the best optimizations on a per method basis within a dynamic compiler. This paper is close to our research as to the approach they have used. The authors used the machine learning technique of logistic regression to derive a model that can determine which optimizations to apply based on the features of a method. While this approach is limited by its inability to change the order of optimizations, it shows that enabling/disabling of different phases for certain methods achieves significant speedup for most of the benchmarks. This speedup for a dynamic compiler may come from removing the compiler overhead with the same or nearly the same quality of the generated code or from improving the generated code. For optimizing Java methods, it was possible to search over the entire space of optimizations, at least in the case of optimization levels O0 and O1. However, if the number of optimizations is large, it is necessary to chose a set of randomly generated settings.

The authors conducted two case studies to evaluate their approach. They provided both the running time and the total time to show what component was speeded up (the generated code, the compile time or both). The first was used on the SPECjvm98 benchmark and the O0, O1 and O2 levels of optimization and produced a speedup of 4%, 3% and 29% respectively. The second one, based one the DaCapo+ benchmark, showed even better result, 33% for O2 level. However, it is not clear whether the random approach will work in the case of huge search space and when the optimizations are used rarely. The difference with our approach is that it is possible only to enable/disable certain optimizations. We tried to be more flexible that allowed us (i) to change several heuristics inside the transformation and (ii) to work with non-binary heuristics.

Cavazos *et al.* [5] tried to determine the best setting for a group of compiler optimizations inside the static open-source commercial compiler called PathScale EKOPath [7], which is a non-trivial problem due to the non-linear interaction of compiler optimizations. In this paper several techniques were developed to search the space of compiler options to obtain a good solution, but most of them were time-consuming. This paper proposes a new approach based on performance counters as a useful tool for finding good compiler optimization settings. The authors used the performance counter values collected from a few runs of the program as input to an automatically constructed model which outputs a probability distribution of good compiler optimizations to use. This model examines performance counter values of a new program and, using prior knowledge from previously examined programs, determines the optimization setting most likely to result in a speedup and improved performance counter values. One of the main advantages of measuring with performance counters is that they do not affect the running program, and, hence, do not influence overall performance.

This paper showed that the run-time information (e.g. performance counters) can be very powerful in describing a method, and, clearly, can be used to improve our approach results.

6.3 Iterative Compilation

Several works on iterative compilation, a popular approach for optimizing programs, which is based on a repeated compiling and reevaluation with different options, appeared in recent years ([1, 10]) that tried to apply machine learning algorithms and solve practical questions inside compiler infrastructure [10].

Agakov *et al.* [1] developed a new methodology to reduce the number of program executions, which is a traditional problem for iterative compilation. They evaluated two models based on program features: independent identically distributed and Markov model. The authors applied them to two embedded systems with the UTDSP benchmark, however this benchmark had programs with sizes that ranged from 20-500 lines of code and runtimes below 1 second, making them much simpler to analyze than SPEC [8] benchmarks. The number of transformation that were used was a dozen of classical optimizations like loop unrolling, dead code elimination, copy propagation, and common subexpression elimination [12]. Their models could speed up iterative search by an order of magnitude. This resulted into an average speedup of 1.22 to 1.27 depending on the platform.

A speedup of the iterative search is also a serious problem in our approach while gathering training data optimal values. Hence, the ideas from this work can be useful for decreasing the number of evaluations while searching over a large space of heuristic values.

Fursin *et al.* [10] proposed a practical way to modify a current compiler, allowing external frameworks to change internal compiler decisions. The authors mentioned a set of characteristics that should present in a good compiler framework. Most of them are present in our approach: reuses for the compiler program analysis, a simple and unified mechanism to obtain information about compiler decisions, transparency for user (at high level), fine-grain (function level) and only legal transformation for a given application. These characteristics are a good starting point for creating a new compiler or modifying an existing one.

In this chapter we described the research papers that were closed to our research and provided the necessary basis for our work. We mentioned both ”single” and ”multiple” heuristic papers, and provided some information about iterative compilation research results that defined our future work direction.

The next chapter will provide conclusions of this research paper and discuss its future potential.

Chapter 7

Conclusions and Future Work

Any road followed precisely to its end leads precisely nowhere.

— Frank Herbert

Compiler writers have always had to deal with complex systems that are hard to model. As we mentioned in Chapter 1, they need to solve NP-hard problems efficiently and heuristics are unavoidable in this case. However, this means that we need to find the optimal values for these heuristics, sometimes for a set of different architectures and, clearly, different applications. Compiler developers can generate highly effective heuristics themselves, but the number of person hours that may be necessary to find the best parameters for these heuristics may be prohibitive.

Our research experimented with the automatic search for the best heuristic values using machine learning techniques. As an application of our ideas we chose the problem of decreasing the compile time while preserving or improving the execution time in a complex commercial compiler. The reasons were described in Chapter 2.

We developed a framework for the TPO compiler and applied the logistic regression technique to predict valuable transformations. The learned classifier can be effective in predicting which transformations are needed. Experiments showed that we can decrease the compile time by at least a factor of two with a negligible increase in the execution time (1%). However, in the case of misprediction the increase can be significant. We also found several bugs, both correctness and performance. The last ones can be the reason for reevaluating certain heuristics and/or transformation decisions.

7.1 Future Work

In this research we were more focused on decreasing the compile time, however, it is clear that the execution time is the primary goal in many cases. Hence, the extension for our approach to be applicable for this problem seems a first natural step. We are also interesting in trying other machine learning techniques (e.g. support vector machines or neural networks) which may have better prediction quality. Clearly, the training data set should be extended by other benchmarks and applications if we want our classifier to be more general.

We believe that our technique is general enough to be a suitable choice for tuning both static and dynamic compilers (e.g. just-in-time compilers). We even expect it to work better in the case of dynamic ones, because we will get a better description during runtime using the same features, and other features can be added if needed.

We want to add the dynamic features (e.g. by using performance counters) to increase the quality of a method description. The reason for this is that the usefulness of static features depend on the current input of your benchmark. For example, a small array size may significantly decrease the value of ”memory” transformations. But later the same method can be used with another input where these transformations will be necessary. Dynamic features allow us to distinguish these two cases and can increase the prediction quality.

We believe that our framework can help engineers and compiler developers to deal with the time-consuming heuristic tuning for new transformations, architectures, and for specific applications. It can also motivate compiler writers to a certain redesign in the existing compiler.

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] A. Asuncion and D.J. Newman. UCI machine learning repository. In *http://www.ics.uci.edu/∼mlearn/MLRepository.html*, 2007.

[3] Christopher Bishop. Neural networks for pattern recognition. *Oxford University Press*, 2005.

[4] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, 1997.

[5] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO '07*, pages 185–197, 2007.

[6] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06: Proceedings of the 21st annual ACM*

SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 229–240, New York, NY, USA, 2006. ACM Press.

[7] PathScale EKOPath Compilers. In *http://www.pathscale.com/*.

[8] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmarks. *http://www.spec.org/cpu2000/*, 2000.

[9] Open Research Compiler for Itanium Processor Family. In *http://ipf-orc.sourceforge.net/*.

[10] Grigori Fursin and Albert Cohen. Building a practical iterative interactive compiler. In *1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference*, pages 1–14, January 2007.

[11] MATLAB. In *http://www.mathworks.com/*.

[12] Steven Muchnick. Advanced compiler design and implementation. *Morgan Kaufmann*, August 1997.

[13] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, pages 78–86, New York, NY, USA, 2004. ACM.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Neurocomputing: foundations of research*, pages 673–695, Cambridge, MA, USA, 1988. MIT Press.

[15] Jonathan Richard Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. *School of Computer Science, Carnegie Mellon University*, August, 1994.

[16] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. *Proceedings of the international symposium on Code generation and optimization*, pages 123 – 134, 2005.

[17] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 77–90, New York, NY, USA, 2003. ACM Press.

[18] Arie Tal. Method and system for managing heuristic properties. In *US 20070089104*. US Patent, April 19 2007.

[19] V.N. Vapnik and A.Ya. Chervonenkis. Theory of pattern recognition (in russian). *Nauka, Moscow*, 1974.