IMPROVING COMPILER OPTIMIZATIONS USING MACHINE LEARNING

by

Sameer Kulkarni

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer and Information Sciences

Summer 2014

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

IMPROVING COMPILER OPTIMIZATIONS USING MACHINE LEARNING

by

Sameer Kulkarni

Approved: _____
Errol L. Lloyd, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: _____
James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

John Cavazos, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

James Clause, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Chengmo Yang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Mario Wolczko, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

The work done during my years as a graduate student has been inspired, and enabled by many people. I am grateful for their support and encouragement and wish that I am able to emulate their support and kindness at every opportunity available.

First, I would like to thank my advisor John Cavazos whose support and confidence was absolutely instrumental, and evident at every step of this dissertation. His patience and kindness provided me with the confidence to continue during my most troubling times, in research as well as personal life.

I want to thank Dr. Mario Wolczko for his inputs and encouragement during my summers at Oracle, and the opportunities provided that helped me during this research, Dr. Christian Wimmer and Douglas Simon for their, insights and the freedom I received from them work on some exciting projects. I am grateful and appreciate the opportunity offered to me by Elenita Silverstein at JPMC that helped me immensely in writing the final chapter of this thesis.

I would also like to thank all the present and past members of Cavazos Lab for the provided help, offered friendships, and the sense of belonging above all. I would also like to thank my friends from before and during grad school, who helped in helping me proof read, find and correct a lot of typos and errors. I would like to show my gratitude with copious amounts of C_2H_5OH, when we meet.

Finally and most importantly I would like to thank my family, my mother Urmila Kulkarni and father Col. S. A. Kulkarni, my sister Anagha and my wife Rasika, for their support and molding me as a person I am today. I would not be here had it not been for your encouragement and support.

Dedicated to:

My late mother,

wish you were with us today...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The increase in the use of computation in every walk of life and its pivotal role in creating new avenues that were not possible before computers is obvious and does not warrant proof. This dramatic rise in the use of computers would continue for the foreseeable future. The hardware has continued to grow in complexity, and programming languages have evolved to make it easier for a human being to design and construct more and more complex solutions. The only bridge between the ever increasing complexity of the underlying hardware and the increasing simplicity of the language designed to describe the solutions that run on it is the *compiler*.

Compilers have grown more and more complex in the past, and will continue to grow in functionality and complexity in two primary directions. One direction is in the ability of the compiler to increasingly understand languages that are closer and closer to human language, that would drastically reduce the human effort involved in designing a solution. The second direction, that is also the primary focus of this dissertation is in the ability of the compiler to create more and more complex machine code that can not only run on the underlying hardware, but also take advantage of the hardware's ever increasing complexity in improving application performance. This role of the compiler, to translate the original source code into complex machine code and squeeze all possible performance from the hardware can be quantized into discrete steps. These discrete steps that change and transform the code with the goal of optimizing the final machine code are referred to as *optimizations*.

The number of optimizations that are available in modern day compilers are in their hundreds, and would only grow in number in the future. This increase in the number of optimizations available to the compiler is primarily due to the fact that each

optimization would try and target specific code constructs and increase their efficiency by applying specific templates. As the optimizations target increasingly specific code constructs, increasing number of optimizations that are being applied to a specific code may have never been intended for the code that is presently being compiled. At present all compilers apply a given set of optimizations blindly to all the code being compiled by the compiler, with the assumption that the optimization would either increase the performance of the code or make no changes. This assumption would mean that either the optimizations are not aggressive enough to reach their full potential for fear of having unintended consequences, or mean that in certain cases even degrade code performance. Knowing which optimizations to apply from the hundreds available in modern day compilers becomes difficult and important at the same time. Our research in improving compiler optimization planning aims to solve this impasse.

Selective application of the optimizations from the vast number of available optimizations to the compilers is the key to increasing performance of the code being compiled. This selective application of the optimizations either involves extensive human involvement that would be (and probably already is) too complex and unsustainable or involve the use of intelligence embedded in the compiler itself. Performance improvement achieved using compiler optimizations without any input from an application developer provides this crucial boost to the application with no recurring associated cost. Another advantage of such techniques is that it can provide performance improvement over and above an already optimized code design.

The process of providing the compiler with such intelligence is the primary focus of this dissertation. The optimization plan of the compiler can be primarily divided into three different logical parts, *Optimization Selection*, *Optimization Ordering*, and *Optimization Tuning*; we would present our research and specific solutions to improving all three of these steps involved in optimization planning.

Optimization Selection is the processing of selecting the beneficial optimizations from

a given set of optimizations. We present an optimization selection solution on the widely used GCC compiler used to compile a financial library. The financial library is used by one of the largest commercial financial firms to model their risk exposure, and the computational needs of this financial library are a significant chunk of the total computation needs of the firm. We achieve a speedup of 2% to 4% compared to the baseline, which could (if implemented) directly impact their extremely large annual computational budget.

Optimization Ordering is the process of arranging the order in which a set of optimizations are applied to a given piece of code. The application of multiple optimizations modify and transform the codes that they are applied to, and thus indirectly interact with each other, some of the optimizations are clean up optimizations and others could be enabling optimizations for another set of optimizations. Such interdependence of optimizations makes it more and more difficult to understand intuitively the right order to apply these optimizations, and generates the need for a method to generate good *optimization orderings*. This problem is usually referred to as phase ordering, and has been considered to be a difficult problem to solve in the past. In this dissertation we show that using source feature analysis in combination with machine learning algorithms can provide us with a robust heuristic in solving phase ordering. We use the Jikes RVM (Jikes Research Virtual Machine developed by IBM) to present our results and achieve a 4% to 8% improvement in the final performance of a given set of benchmarks.

Finally *Optimization Tuning* is the process of tuning a single optimization to improve its efficacy. The example of optimization tuning that we study in this dissertation is method inlining, We use the Java HotSpot Virtual Machine (the most commonly used Java VM developed by Sun Microsystems and now actively developed and maintained by Oracle) and Maxine VM (A Java Research VM developed by Oracle) to present our results in improving the performance of method inlining, and achieve a 10% to 14% improvement in performance. We also show an interesting twist to

presenting the final machine learning heuristic in the form of a decision tree, and discuss the advantages of using this over an artificial neural network.

In presenting our research on Optimization Ordering and Optimization Tuning we use dynamic compilation environments. In a dynamic compilation environment, the advantages of improving any aspect of the compilation process are compounded as the compilation of the code is performed in parallel with the application being executed. Any reduction in the compiler burden would make more resources available for the executing application, and thus provide a further boost the final performance of the application. Another interesting study during this research has been in the use of source code features as a way to discretize and characterize the code being compiled in the form of a vector, that can be used by the machine learning algorithm as an input to propose customized recommendations to improve the compiler optimization plans. We feel that using source code features to characterize the code being compiled is crucial in the next step of evolving the compiler into providing intelligent and customizable solutions in the future.

Chapter 1

INTRODUCTION

For a long time increasing application performance has primarily relied on increasing clock frequencies, and no other change in the underlying code or any major change in the architecture. This *free ride* is no longer possible due to energy constraints. Recent advancements in computer architecture have focused on using multiple cores to distribute computational workload instead of increasing the speed of a single processor. Taking advantage of the more recent advancements in architecture requires a concerted effort on the part of the architecture designer as well as the application developer. This effort is required as there are more processing cores that are available to perform parallel computation, and there could be multiple applications that are competing for a limited set of resources. In such a situation of resource contention and multi threaded architectures, the role of a compiler becomes more important.

In order to be able to optimize all kinds of codes being compiled by the compiler, the compiler writers have added many compiler optimizations. Modern day compilers have consistently added more and more optimizations for example GCC has more than 200 such optimizations, JikesRVM applies more than 150 total compiler optimization steps. These compiler optimizations help to increase the performance of specific snippets of code. However these optimizations often may not increase performance of large portions of codes that they are applied on, this would mean wasting compiler computation time and resources. In some cases the optimization might also degrade the performance of certain snippets of code being compiled, in such situations not only are the resources of the compiler wasted, but also the final code would perform worse that it would have had the compiler not applied the specific optimization. Given the

number of optimizations available it is not possible for an application developer or even the compiler writer to create an exhaustive set of scenarios that could benefit all possible code combinations that one would encounter in the real world. In order to mitigate this problem it is imperative that there be a system that can intelligently create, modify and curate the optimization plan specific to the code being optimized, and not try have have a globally static plan with the aim to optimize any code that can possibly be generated.

In this dissertation we use three different Java Virtual Machines (JVMs) and the GCC compiler (version 4.8). During the rest of the thesis we refer to the VMs as the compiler, however it should not be confused with the Java Compiler that converts the Java source code to Java Bytecode. The reason for this is that the VM takes in the Bytecode as the source code and either uses an interpreter to execute the code directly or uses the in-built compiler to compile the code into native code and run it directly on the host. In these experiments and others in which we use the term *compiler* to refer to the compilation that happens in the VM when the bytecode is converted to native machine code, this is the compilation step that we would try to tweak, optimize and improve.

The primary area of research in this dissertation is the use of source feature analysis and machine learning in the use of improving the application of compiler optimizations. Compiler optimizations aim at improving the quality of the final code being generated. This chapter gives a brief introduction to compiler optimizations and it's importance, and then talks about the difference between optimization selection, optimization tuning, and phase ordering.

1.1 Motivation

Compilers read in source code written by an application developer and translate it into machine code. During this process of translation the source code goes through multiple iterations of small translations also called transformations. Some of these

transformations are required and mandatory for the correct generation of the final machine code, and some are optional. The goal of these optional transformations is to improve quality and performance of the code being generated, the improvement in quality could be in the form of reducing the running time of the code or reducing the memory footprint of the code during execution. These optional transformations are also called *compiler optimizations*. Most compilers apply many compiler optimizations, and they apply these *compiler optimizations* one at a time. The advantage of compiler optimizations are that they can improve efficiency and code performance over and above the optimized code written by the application developer. This improvement through compiler optimizations could be due to the fact that some of the optimizations become possible during compile time, or the writer may have written some code that could be made more efficient.

In any modern day compiler that is extensively used the compiler might see a very large amount of source code. Since this code is written to perform a wide range of tasks and by a large number of different people, the code that is being compiled might look very different from one application to the other. Thus in order to be able to compile and optimize such a large variety of source code the compiler has in its arsenal a large number of optimizations. However in certain situations a compiler optimization might leave the application performance unaffected, or even degrade it in certain circumstances. Another way to put it would be to say:

The compiler optimization can improve the performance of some of the code all of the time or all of the code some of the time, but not all of the code all of the time.

Compiler optimizations by definition maintain the correctness of the code being compiled, but provide no guarantees on the efficiency of the code. If the compiler is smart it might be able to apply the right set of optimizations that have a positive effect on the code being compiled. The method of selecting the right set of optimizations from a given set of available optimizations has been an active area of research for multiple decades and there are still more questions that need to be answered. The ability to

apply only beneficial optimizations would not only benefit the code being compiled, but would also reduce the load on the limited resources of the compiler.

Since optimizations are applied one at a time the order in which they are applied also affects the performance of the final compiled machine code. Fixing the order in which these optimizations are applied is a difficult problem and is called phase ordering.

There are three fundamental challenges in most problems that involve compiler optimizations, *exhaustive exploration*, *sparse search space*, and *performance evaluation*. Any solution that attempts to solve or mitigate a compiler optimization plan would need to address these three challenges. In section 1.1, we take an example of each of these challenges and steps required to mitigate them.

Exhaustive exploration

The number of optimization configurations that generate accurate code is extremely large. It is not possible for a compiler or the compiler writer to go through and test out all possible configurations. An easy way to understand the scope of the size of the search space is when we consider the example of phase ordering. As a standard practice compiler designers set a limit on the number of optimizations applied during a specific compilation. For example in the Jikes RVM compiler the number of optimizations that are applied during O3 optimization level are 67. If we consider that there are only 40 optimizations that can be applied and our phase ordering is limited in length to a maximum of 25 optimizations long, then the number of possible optimization orderings would be 40^{25}. Normally to measure the performance of the compiler optimization sequence, one would have to compile the source code and then run the generated machine code, and measure the running time. To compile and run a piece of code might take a few seconds. If the time needed to execute this code is reduced by nine orders of magnitude, and take only one nano second to evaluate each optimization sequence, total time required to evaluate all possible sequence would still

be more than million times the age of the universe.[1] The rough estimate of the size of the search space has been calculated in the Table 1.1 below:

Rough Calculations [1]	
Number of optimizations available for phase ordering = ops	$= 40$
Sequence length of the final phase ordered sequence of optimizations = len	$= 25$
Time to compile and run a piece of code = 1 n sec.	$= 10^{-9}$ sec
Number of unique sequences = ops^{len}	$= 1.125 * 10^{34}$
Time take to evaluate one sequence	$= 1 * 10^{-9} sec$
Total time for all evaluations	$= 1.125 * 10^{25} sec$
Age of the universe	$= 4.354 * 10^{17} sec$

Table 1.1: Table calculating the enormity of phase ordering search space

From the Table 1.1 we can see that the amount of time needed is closer to million times the age of the universe. The Table 1.1 assumes that we use just one fixed sequence on all the methods of a benchmark, and past research has shown that each method would perform best with individually customized sequence of optimizations. This means that the number of possible combinations would be even larger in the real world. Thus brute force cannot and never will be an option to iterate the search space of all available phase orderings. Due to the extremely large size of the search space, there have been some attempts to find alternate methods to intelligently prune the search space, and try and make this problem more manageable.

Sparse search space

The previous section gave a rough picture of how big the search space could get for even small scale problems. Another problem that is common is how sparse the search space could be. Sparse search space primarily refers to the concentration of good optimization configuration points in the total possible search space. If there are

[1] The values in this table are an pseudo realistic examples based loosely on the compilers being used. The time taken to evaluate one sequence on one benchmark is an almost unrealistically optimistic value and would certainly be much larger in the real world.

a lot of good optimization points in a given search space, then the compiler could just stumble upon a good configuration fairly quickly by just sampling the search space randomly. However if the search space is fairly sparse, or that the number of good optimization configurations are few and far in between, then it would make it more and more improbable for just random sampling to stumble upon a good configuration. If the compiler were to exhaustively step through each possible optimization configuration, to check its effectiveness, a vast majority of these tested optimization configurations would have to be discarded. This is based on the set of experiments that were performed using a random method inliner.

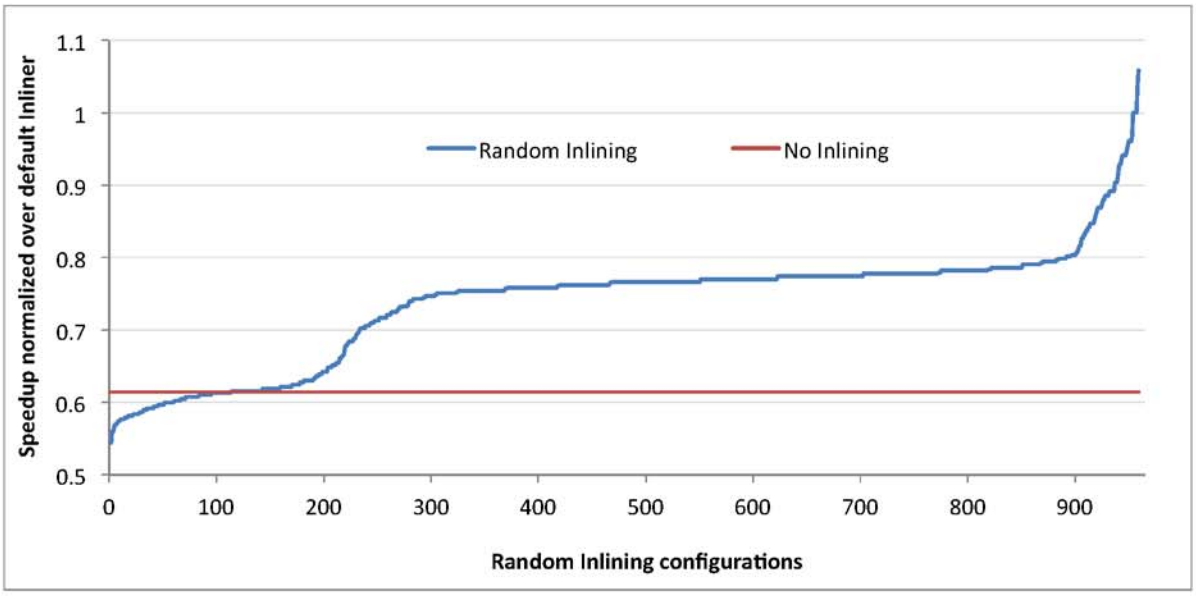


Figure 1.1: The graph above shows the random inliner

The graph in Figure 1.1 shows a set of experiments where a random inliner was used to make method inlining decisions. The random inliner was used in place of the default inliner to gain a better understanding of the search space in method inlining. Here instead of using the default method inliner to make the inlining decision we used a random number generator to generate a random number that governed the inlining decision. This random inliner was used over a thousand times to make inlining decisions. Of the 1000 times that the random inliner was used, only two

specific instances was the random inliner able to match or improve on the default inliner. This means that a vast majority of sampling points in the search space of possible inlining decisions make a worse decision than the default inliner. The red line in the Figure 1.1, shows the speedup that one would achieve if the compiler completely disabled the method inlining. There are almost 170 points that are equal to or worse than the performance of the code if there was no inlining at all, and only two points that were better than the default inliner. These numbers present an interesting result, that the compiler is almost a hundred times more likely to find an optimization point that is worse than or equal to disabling the method inlining completely, than find a point that is as good as or better than the default inliner. This result could give us the rough picture of how futile it would be to perform random sampling as compared to an intelligent search approach.

Performance prediction

A fundamental problem commonly faced when trying to quantify the benefits of a compiler optimization is the difficulty in predicting its effects on a piece of code. The most accurate method to quantify the effects of a particular optimization plan is to first compile the code and then actually execute it to collect the timing information. This makes the testing of a given solution very time consuming, and also makes a lot of machine learning techniques unsuitable.

1.2 Compiler Tuning

There are primarily three different ways of improving the performance of compiler optimizations for a given piece of code:

- Optimization Phase Ordering
- Optimization Tuning
- Optimization Selection

A very brief introduction to the above three ways is presented below. In order to understand these methods we use an example of an imaginary compiler C, that has 26 compiler optimizations named A to Z.

Optimization Selection

Optimization selection is the process of selecting the set of optimizations that have the best chance of improving the quality of the code. For example given a piece of code being compiled by C the Optimization selection algorithm might pick a subset of the optimizations available to it, and only apply optimizations B, D, E, and K. Note that there is no fixed order that is being dictated by the algorithm itself.

Optimization Ordering (Phase Ordering)

In Optimization Ordering the algorithm would select the optimizations to be applied, and also select the order in which the optimizations are applied. For example, if the compiler was to apply optimizations B,D,E, and K, the phase ordering algorithm would control the order by applying optimization E, then applying optimization K followed by B and D ($E \rightarrow K \rightarrow B \rightarrow D$). This ordering would potentially give different results than applying the optimizations in a different order (e.g $D \rightarrow B \rightarrow K \rightarrow E$). Optimization ordering is important as some optimizations are clean up optimizations, only effective when another optimization has already been applied to the code already. This is a tougher problem to solve and has been the primary area of study during our research.

Optimization Tuning

Some compiler optimizations are binary optimizations, where the only choice the compiler has over them is to either apply the optimization or to not apply the optimization. Other optimizations are more complex optimizations and they can be finely tuned. Optimization Tuning is the process of modifying and restructuring the parameters of a mutable optimization in order to increase its efficacy. For example, if optimization K is the optimization that has multiple parameters that govern its

behavior, and is being tuned, the compiler could try multiple settings and levels to try and find the values that provide the best performance improvement.

1.3 Optimization Ordering

There has been very little effort done on improving the phase ordering problem. The research during the initial stages of this dissertation aimed at providing a good solution to compiler phase ordering.

Importance of phase ordering

As stated earlier in Section 1.2, the compiler optimizations interact with each other. These interactions can be basically be classified into three different kinds, namely:

Enablers Some optimizations enable other optimizations or create the environment required for another optimization to execute.

Cleanup An optimization is a clean up optimization if it needs to be applied after a particular optimization or more commonly after a set of optimizations.

Grouped Optimizations Set of optimizations that need a starting optimization and end with an ending optimization. All these optimizations in the group must be applied only between these two starting and ending optimizations, thus can be called grouped optimizations.

These dependencies need to be respected during the design of the compiler, and also when the compiler is generating an optimization sequence. *Phase-ordering* has been an open problem in compilation research for decades. The only way to know the effectiveness of a sequence is to apply the optimizations to the code and run the code either in the simulator or on the actual machine. Another problem is that the number of possible orderings make it impossible to use brute force on these problems.

Solving the problem of *phase-ordering* of optimizations has been approached in many ways. Compiler writers typically use a combination of experience and insight to construct the sequence of optimizations found in compilers. In this approach, compromises must be made, e.g., should optimizations be included in a default fixed sequence

if those optimizations improve performance of some benchmarks, while degrading the performance of others for example, GCC has around 250 "passes" that can be used, and most of these are turned off by default. The GCC developers have given up in trying to include all optimizations and hope that a programmer will know which optimizations will benefit their code.

In optimizing compilers, it is standard practice to apply the same set of optimizations in a fixed order on each method of a program. However, several researchers [3, 4, 5], have shown that the best ordering of optimizations varies within a program, i.e., it is function-specific. Thus, we would like a technique that selects the best ordering of optimizations for individual portions of the program, rather than applying the same fixed set of optimizations for the whole program.

This research develops a new *method-specific* technique that automatically selects the predicted best ordering of optimizations for different methods of a program. We develop this technique within the Jikes RVM's Java Hotspot compiler to automatically determine good phase-orderings of optimizations on a *per method* basis. Rather than developing a hand-crafted technique to achieve this, we make use of an artificial neural network (ANN) to predict the optimization order likely to be most beneficial for a method. Our ANNs were automatically trained using *Neuro-Evolution for Augmenting Topologies* (NEAT) [6].

We trained our ANNs to use the source features of the method being compiled as input properties, its current optimized state and to output the optimization predicted to be most beneficial to the method at that state. Each time an optimization is applied, it potentially changes the properties of the method. Therefore, after each optimization is applied, we generate new features of the method to use as input to the ANN. The ANN then predicts the next optimization to apply based on the current optimized state of the method. The technique solves the phase-ordering problem by taking advantage of the *Markov property* of the optimization problem. That is, the current state of the method represents all the information required to choose an optimization to be most

beneficial at that decision point. We discuss the Markov property and our approach in more detail in Section 2.6.

The application of machine learning to compilation has received a lot of attention. However, there has been little effort to "learn" the effect that each optimization has on the code and to use that knowledge to choose the most appropriate optimization to apply. To the best of our knowledge, the technique described here is the first to automatically induce a heuristic that can predict an overall optimization ordering for individual portions of a program. Our technique learns what order to apply optimizations rather than tuning local heuristics, and it does this in a dynamic compilation setting. This approach can provide performance improvement for even a well engineered system.

1.4 Optimization Tuning

Optimizations cannot always be considered as binary functions. Some optimizations can change in aggressiveness or modify their behavior based on certain factors or changes to the environment. The environment in this case comprised of the code being compiled and the target machine.

Method inlining is one such optimization that can be extensively tuned. Method inlining is also one of the optimizations that has the largest impact on the performance of the application being compiled. Research done during this dissertation extensively studies this optimization and shows the different parameters that may affect its efficiency.

The Java Hotspot compiler method inlining optimization happens during bytecode load phase of the compilation. The bytecode is loaded when compilation of the class is triggered. The triggering mechanism can be specifically tuned, but is usually after a predetermined number of method invocations of the method. During bytecode load the compiler reads the bytecode and may recursively invoke method inlining of methods that are being called in the method being compiled. Since the method might

have been executed multiple times in the past, these execution scenarios could be made available to the compiler to make a better prediction of the relative importance of each part of the method. It is the presence of this information that can be taken advantage of by the compiler to make better decisions.

Compilation of Hot methods

The method inlining optimization is studied under the Java Hotspot server compiler. All code being executed is initially interpreted and only the codes that are frequently executed are sent for compilation. This method primarily helps in reducing initial compile time and reduces load on the compiler. Another secondary advantage in the case of compiler optimizations is in terms of profiling information. When the *Hot Methods* are actually compiled there is a vast amount of profiling information that is available about the method, and this information can be used by the compiler to more effectively apply the right set of transformations.

1.5 Structure of the dissertation

This thesis has been divided into six chapters. Chapter 2 discusses the related work that is relevant to the research presented in this dissertation, followed by an introduction to Machine Learning and the theory supporting the methods that were used during this research. In Chapter 3 we introduce the concept of *Optimization Ordering*, the need for phase ordering, our proposed solution, the theory that would support our solution and the results that we observed using our methods. The next chapter on method inlining (Chapter 4) discusses a method of tuning a single optimization to extract the best possible performance from applying the optimization. This chapter also introduces the concept of using profiling information in combination with source features to get good performance improvements in method inlining. We also present a method to convert an Artificial Neural Network into a decision tree, to improve the

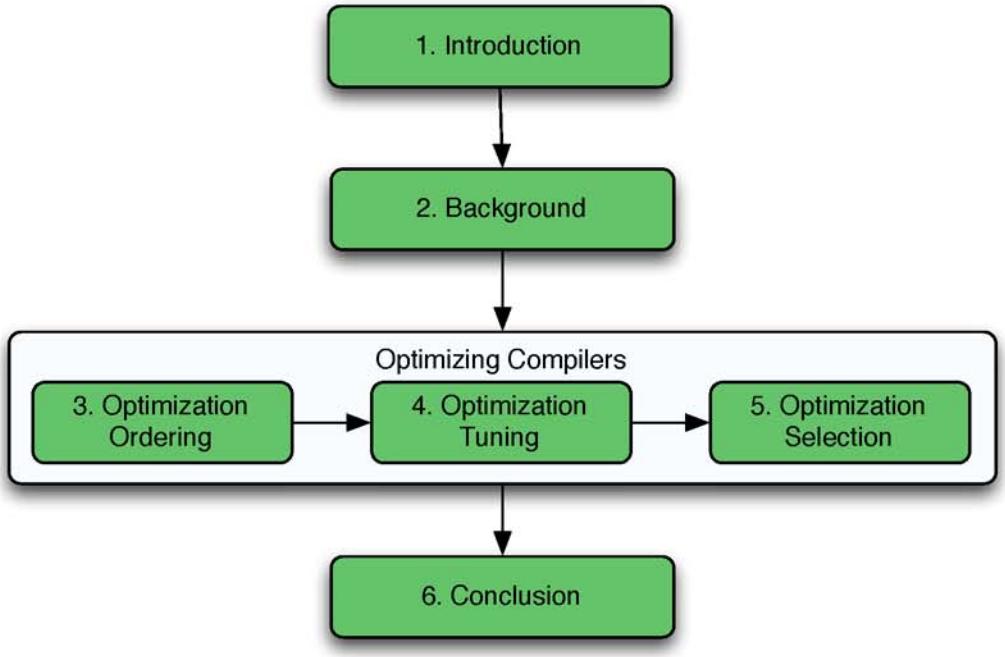


Figure 1.2: Thesis structure.

readability of the final proposed heuristic. In Chapter 5 we present the research conducted with the goal of improving the optimization selection process in a statically compiled environment. We improved the optimization selection process in the GCC compiler to compile a heavily used financial library, followed by our conclusions in Chapter 6.

Chapter 2

BACKGROUND AND RELATED WORK

Compiler optimizations have existed ever since the first compilers that compiled FORTRAN code. Compiler optimizations are in a way the very foundation stone of the effort to develop compilers. As the size and the complexity of the applications grew it became more and more logical to start writing code in a higher level language instead of the basic but extremely difficult to read machine code. This need for compilers can be understood from the point of view of the large projects that were becoming more and more common, however the adoption and the use compilers would only be possible if the compiler could produce code that was comparable in performance to a code that was developed by an application developer using machine code. The direct comparison of the compiled code with hand tuned machine code has been the primary reason for developing empirical rules and specific templates that the compiler would use to improve the performance of the application being compiled by the compiler. These early templates and empirical rules were the first compiler optimizations developed with the aim to generate code that was better than the one written by the application developer.

Since that time more and more compiler optimizations have been introduced. The introduction of a compiler optimization could be primarily for two reasons. The first, to take advantage of a specific feature in the underlying architecture like instruction pipelining, cache or the presence of special registers. The second, to improve small pieces of code that the compiler comes across that can be directly be converted into a more optimal machine code. The number of different architectures and the amount of computational power available have exponentially increased following the Moore's

law, so has the amount and kind of code increased due to the wide spread adoption of computation in all walks of life. This has led to an explosion in the number of optimizations available or needed. Compiler optimizations can be viewed as a two sided coin. One side is the research into creating a new optimization that improves a particular type of code snippet. The other side is to study the already available list of optimizations and measure their suitability. This chapter describes different techniques that have been used by researchers in the past for trying to improve the effectiveness of the optimizations already available.

2.1 Auto-tuning

Auto-tuning is an area that is closely related to this proposal and the study of automatic code generation and optimization for different computer architectures. This technique has been used in many optimization scenarios. Work presented in Li *et al.* [7] shows a way to use genetic algorithms to partition stored contiguous data using a hybrid sorting technique to improve sorting performance. A number of library generators automatically produce high-performance kernel routines [8, 9, 10, 11]. Recent research efforts [12, 13] expand automatic code generation to routines whose performance depends not only on architectural features, but also on input characteristics. These systems are a significant step toward automatically optimizing code for different computer architectures. Recently, Ganapathi *et al.* [14] presented some preliminary results on the application of machine learning to auto-tuning for multi-cores. They showed that auto-tuning of stencil codes, with the assistance of machine learning, was able to surpass performance of tuning by a domain expert. The research displays the great potential for machine learning and search in an auto-tuning environment. However, these prior works have all been largely focused on small domain-specific kernels and still neglect exploring the benefits of learning from a knowledge base of previously explored applications and architectures. The research mentioned above also did not tackle the problem of phase-ordering of optimizations.

2.2 Machine Learning Applied to Compilation

Machine learning and search techniques applied to compilation have been studied in many recent projects [15, 16, 17, 18, 19, 20, 21, 22]. These previous studies have developed machine learning-based algorithms to efficiently search for the optimal selection of optimizing transformations, the best values for the transformation parameters, or the optimal sequences of compiler optimizations. Generally these studies customize optimizations for each program or local code segments, some based on code characteristics. The proposed research is motivated by these studies and makes a significant step forward: the compiler will not only use program characteristics, but will also learn to decide the right ordering of optimizations.

Many researchers have also looked at using machine learning to construct heuristics that control compiler optimizations. Cavazos *et al.* [18] used logistic regression to control what optimizations to apply in JikesRVM. However did not attempt to control the order of optimizations and instead only turn on and off optimizations given the hand-tuned fixed order of optimizations. For the SPECjvm98 benchmarks, they were not able to achieve significant improvements for running time under both non-adaptive and adaptive scenarios. The absence of significant improvements might have been because of the fixed-order of optimizations in Jikes RVM had been highly tuned and there was little room for improvement on top of this ordering by simply turning optimizations on and off. In contrast, we achieve good improvements on SPECjvm98 benchmarks by applying method-specific optimization orderings.

Stephenson *et al.*[15] used genetic programming to tune heuristic priority functions for three compiler optimizations within the Trimaran's IMPACT compiler. For one of the optimizations, register allocation, they were only able to achieve on average a 2% increase over the manually tuned heuristic. Monsifrot *et al.*[22] used a classifier based on decision tree learning to determine which loops to unroll showing a few percent improvement on two different machines. This research aims at controlling multiple optimizations available in the compiler.

Agakov *et al.* [23] describe two models to improve the search for good optimization orders to apply to programs. The first model, called the *independent identically distributed model*, produces a probability vector corresponding to the probability that a transformation occurs in a good sequence for a particular program. When optimizing a new program, a nearest neighbor algorithm is used to choose the probability vector of the program in the training set closest to the program to be optimized. This probability vector is then used to choose optimizations for the new program. The second model, called the *Markov model* simply creates a probability matrix where the probability of an optimization being beneficial depends upon the optimizations that have been previously applied. These models were developed to focus the search for good optimization orderings during iterative compilation. Therefore, these techniques suffers from the same limitations as described in Section 3.1. Additionally, these models use simple nearest neighbor algorithms using the characteristics of the original unoptimized code. Therefore, these models do not take advantage of important characteristics of the code as it is being optimized.

Wang *et al.* [24] have applied machine learning to select the best number of threads for a parallel program and to determine how these threads should be scheduled. The authors use neural networks and prior runs of several programs to predict the number of threads for a new "unseen" program, only requiring a few profiling runs of this new program. Results presented show excellent performance compared to two state-of-the-art techniques. We will significantly extend this work by considering the interplay of optimizations to parallelism and by changing the parallelism configuration (e.g., the number of threads) at runtime based on dynamic conditions.

Fursin *et al.* [25] (as part of the MILEPOST project) have integrated machine learning algorithms in GCC to control these optimizations applied. They show good results on three different architectures, compared to random search of optimizations sequences. However, the machine learning algorithms in MILEPOST do not learn good optimization orderings because as the authors state "this requires detailed information

about dependencies between passes to detect legal orders".

2.3 Phase Ordering

Several researchers have looked at searching for the best sequence of optimizations for a particular program [26, 27, 19, 28, 29, 30]. Cooper *et al.* [26] used genetic algorithms to solve the compilation phase ordering problem. They were concerned with finding "good" compiler optimization sequences that reduced code size. Their technique was successful at reducing code size by as much as 40%. Unfortunately, their technique was application-specific, i.e., a genetic algorithm had to be retrained to find the best optimization sequence for each new program. Also, Cooper [19] proposed a technique called *virtual execution* to reduce the cost of evaluating different optimization orderings. Virtual execution consists of running the program one time and predicting the performance of different optimization sequences without running the code again. These approaches give impressive performance improvements, but have to be performed each time a new application is compiled. While this is acceptable in embedded environments, it is not suitable for typical compilation.

Kulkarni *et al.* [30] exhaustively enumerated all distinct function instances for a set of programs that would be produced from different phase-orderings of 15 optimizations. This exhaustive enumeration allowed them to construct probabilities of enabling/disabling interactions between different optimization passes in general and not specific to any program. In contrast, this research characterize methods being optimized; therefore, the techniques described here learn which optimizations are beneficial to apply to "unseen" methods with similar characteristics.

Stephenson *et al.* [15] used genetic programming to tune heuristic priority functions for three compiler optimizations within the Trimaran's IMPACT compiler. For one of the optimizations, register allocation, they were only able to achieve on average a 2% increase over the manually tuned heuristic. Monsifro *et al.* [22] used a classifier

based on decision tree learning to determine which loops to unroll showing a few percent improvement on two different machines. The results in these papers highlight the diminishing results obtained when only controlling a single optimization. In contrast, this research will control numerous optimizations available in the compiler.

Agakov *et al.* [23] describe two models to improve the search for good optimization orders to apply to programs. The first model, called the *independent identically distributed model*, produces a probability vector corresponding to the probability that a transformation occurs in a good sequence for a particular program. When optimizing a new program, a nearest neighbor algorithm is used to choose the probability vector of the program in the training set closest to the program to be optimized. This probability vector is then used to choose optimizations for the new program. The second model, called the *Markov model* simply creates a probability matrix where the probability of an optimization being beneficial depends upon the optimizations that have been previously applied. These models were developed to focus the search for good optimization orderings during iterative compilation. Therefore, these techniques suffer from the same limitations as described in Section 3.1. Additionally, these models use simple nearest neighbor algorithms using the characteristics of the original unoptimized code. Therefore, these models do not take advantage of important characteristics of the code as it is being optimized.

There is also some work on iterative compiler optimizations where the code is optimized based on predictive modeling based on a machine learning algorithms (independent and Markov models) [31, 32] or using some other models to decide on applying or not applying optimizations [33]. Another similar paper by John Cavazos talks about using logistic regression and using it to predict the best optimizations by using code feature extraction on each method [18]. This approach is the most similar to what we hope to achieve in phase ordering.

2.4 Method Inlining

Method inlining has been well known to be one of the most important optimizations that can effect the performance of an application being compiled. This importance has logically attracted a lot of academic as well as industry led research in trying to get the best method inlining applied by the compiler. Applying good method inlining has at the same time been a hard problem to solve. The difficulty of good method inlining fundamentally stems from the fact that each method inlining decision is interdependent on all the other previous and future method inlining decisions. The universal set of method inlining decisions is a directed cyclic graph, however to make it simple to comprehend it could be visualized as a n-ary tree. Each parent in the tree is the caller, the parent method that can call one or more methods, the callee. In this entire tree the compiler would need to collapse the tree in such a manner that the traversal time is reduced from the root to a leaf node. There are two costs associated with the traversal, the size of the node itself as well as the transition from the parent node to the child node, or from the caller to the callee. A good compiler would need to merge the node keeping in mind both these costs.

Cooper *et al.* [34] present evidence that a "one- size-fits-all" for inlining heuristics does not perform well. The authors discretized the search space in order to reduce searching time. They also suggest a way to make inlining adaptive for a given piece of code. However, the technique requires performing their search on every new program being compiled. In contrast, we are generating fast heuristics that can improve the inliner and do not require search. In a much earlier work Cooper *et al.* [35] also perform inlining on numerically intense Fortran benchmarks, however inlining certain critical function calls lead to a degradation in performance. Inlining of certain critical calls lead to poorer subsequent analysis and less effective instruction scheduling which resulted in an increased number of floating-point stalls. This study further emphasizes our point that more intelligent inlining is required and tuning heuristics through empirical search as opposed to imprecise modeling may be beneficial.

Arnold *et al.* [36] represents the inlining problem as a knapsack problem that calculates the size/speed trade offs to make inlining decisions. They use code size and the running time as a measure of the effectiveness of the proposed solution. This paper however does not talk about inlining enabling other compiler optimizations or the effects of each inlining decision with subsequent inlining decisions. They achieve speedups of about 25% on average over no inlining while keeping the code size increase to at most 10%. Dynamically compiled languages may not have a global view of the code being executed, and this makes it difficult to directly apply the results to our environment. Another area of potential problems is the performance degradation due to overly aggressive inlining.

Hazelwood *et al.* [37] describe a technique of using context sensitive information at each call site to control inlining decisions. This information included the sequence of calling methods that lead to the current call site. Using this context sensitive information they were able to reduce the code space by 10%, however this resulted in increase in the running time of the benchmarks. When implementing this approach one must take care in not using too much context sensitivity which can degrade performance. They suggest several different heuristics for controlling the amount of context sensitivity, but there is no clear winner among them. This technique is similar to our tuning process, in using context sensitive information.

Dean *et al.* [38] develop a technique to measure the effect of inlining decisions for the programming language SELF, called *inlining trials*, as opposed to predicting them with heuristics. Inlining trials are used to calculate the costs and benefits of inlining decisions by examining both the effects of optimizations applied to the body of the inlined routine by comparing the present code and environment with past experiences. The results of inlining trials are stored in a persistent database to be reused when making future inlining decisions at similar call sites. Using this technique, the authors were able to reduce compilation time at the expense of an average increase in running time.

This work was performed on a language called SELF, which places an even greater premium on inlining than Java due to its frequently executed method calls. This technique requires non-trivial changes to the compiler in order to record where and how inlining enabled and disabled certain optimizations. We assert that better heuristics, such as the ones found in this paper, can predict the opportunities enabled/disabled by inlining and may achieve much of the benefit of inlining trials.

Cavazos *et al.* [39] presents a way to perform fast search over the possible values used to tune inlining heuristics by using genetic algorithms. The paper presents evidence towards the fact that each code might require different settings and "one-size-fits-all" may not be accurate in fine tuning inlining heuristics. Leupers *et al.* [40] experiment with obtaining the best running time possible through inlining while maintaining code bloat under a particular limit. They use this technique for C programs targeted at embedded processors. In the embedded processor domain it is essential that code size be kept to a minimum. They use a search technique called branch-and-bound to explore the space of functions that could be inlined. However, this search based approach requiring multiple executions of the program must be applied each time a new program is encountered. This makes sense in an embedded scenario where the cost of this search is amortized over the products shipped but is not practical for non-embedded applications.

2.5 Machine Learning

In this section we give a detailed overview of different machine learning approaches used during the course of this research. The three different algorithms that we used were *Artificial Neural Networks(ANN)*, *Genetic Algorithms(GA)*, and *Decision Trees*. We use an evolutionary approach to training the ANN as well as the GA, and use the C4.5 algorithm to train the Decision Tree.

Artificial Neural Network

Artificial Neural Networks fundamentally are weighted directed graphs that have a set of input nodes and a set of output nodes. The Figure 2.1 shows a rough structure of a simple ANN. The green nodes to the left are input nodes that take in parameters as inputs, these inputs are then propagated across the graph in conjunction with the weights of the connection between two nodes and in the end we are provided with a set of outputs. These outputs can then be used by the engine to make or perform decisions. The internal nodes that are neither input nodes nor output nodes are called internal nodes and are used encode and emulate complex behaviors. The graph could possibly have cycles that represent a cyclic dependencies of a set of parameters on each other, however in our set of experiments we discourage the generation of ANNs that have cyclic dependencies to reduce the computational complexity of the final ANN that is generated after training.

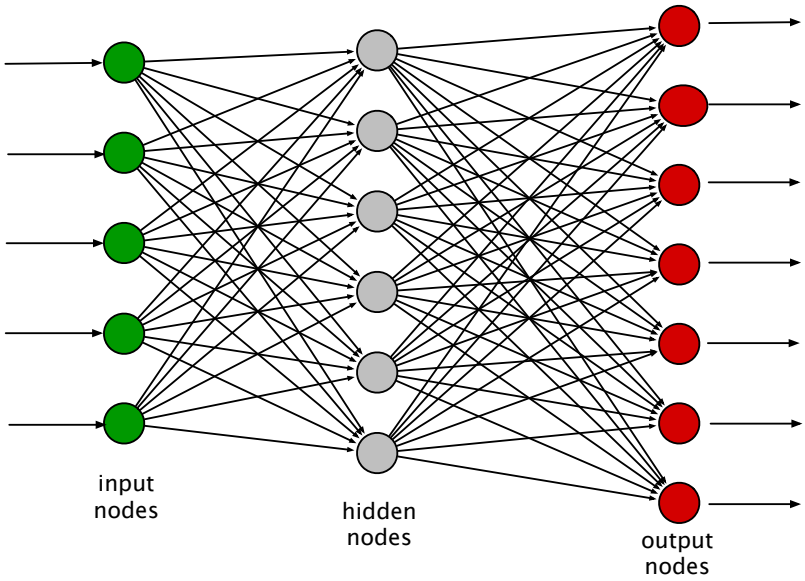


Figure 2.1: General structure of an Artificial Neural Network

During the course of this research we train ANNs in an evolutionary manner using an evolutionary engine named NEAT (*Neuro Evolution of Augmented Topologies*) explained in more detail in Section 2.8. We used the trained ANN to control the phase

ordering in JikesRVM and the method inlining decision in the Java Hotspot VM and the Maxine VM.

Genetic Algorithm

Genetic algorithms are used in this research to compare the effectiveness of GA in performing phase ordering and optimization tuning in the form of method inlining. GAs are also used to tune GCC in improving the performance of the Financial Library. A disadvantage of using GA is the fact that it can not take into account any inputs, and thus cannot be used to emulate a system that would need to change and adapt dynamically.

Decision Tree

We use the decision trees in order to increase readability of a machine learning heuristic. The decision tree is generated using the C4.5 algorithm and used to make method inlining decisions. The method of generating the decision tree is presented in more detail in Section 2.9.

Primarily discussed here in this research is a neuro-evolutionary approach to training neural networks, learning is used to construct a good optimization heuristic for the optimizer within the compiler. In the next section we describe the Markov Property, and our reasons for modeling our solution in a way that it posses the Markov property. The Section 2.7 outlines the different activities that take place when training and deploying a machine learning heuristic. This is followed by Section 2.8 describing how we use NEAT to construct an ANN, how we extract features from methods, and how these features and ANNs allow us to learn a heuristic that determines the order of optimizations to apply. Figure 3.3 outlines our technique.

2.6 Markov Property

Most compilers apply optimizations in a fixed order, and this order is tuned for a particular set of benchmarks. This tuning process is performed manually and is tedious and relatively brittle. Also, the tuning procedure needs to be repeated each time the compiler is modified for a new platform or when a new optimization is added to the compiler. Most importantly, we have empirical evidence that each method within a program requires the application of a specific order of optimizations to achieve the best performance. This research proposes to use machine learning to mitigate the compiler optimization phase-ordering problem.

Determining the correct phase ordering of optimizations in a compiler is a difficult problem to solve. In the absence of an oracle to determine the correct ordering of optimizations, we must use a heuristic to predict the best optimization to use. A drawback of training a neural network that can perform phase ordering is the added difficulty if the ANN would need to remember state. If the ANN is asked to remember the last applied optimization, the compiler designer might want the ANN to remember two previously applied optimizations. There is no limit to the number of optimizations that the ANN might need to remember. In such a situation it is not possible to create an elegant solution.

We formulate the phase-ordering problem as a *Markov Process*. In a *Markov Process*, the heuristic makes a decision on what action to perform (i.e., optimization to apply) based on the current state of the environment (i.e., the method being optimized). In order to perform learning, the state must conform to the *Markov Property*, which means that the state must represent all the information needed to make a decision of what action to perform at that decision point. In our framework, the current state of the method being optimized serves as our Markov state because it succinctly summarizes the important information about the complete sequence of optimizations that led to it.

2.7 Overview of Training and Deployment

Our experiments depend on two distinct phases, training and deployment. Training occurs once, off-line, "at the factory" and is equivalent to the time spent by compiler writers designing and implementing their optimization heuristics. Deployment is the act of applying the heuristic at dynamic compilation time to new "unseen" programs.

Training at the factory

All the training of the compiler and the ANN or other Machine learning algorithm presented during this research is done at the very beginning during the compiler construction phase. The compiler construction happens at the *factory* where once the basic compiler is developed the compiler developers would tune the compiler to improve the quality of the code that is produced in the end.

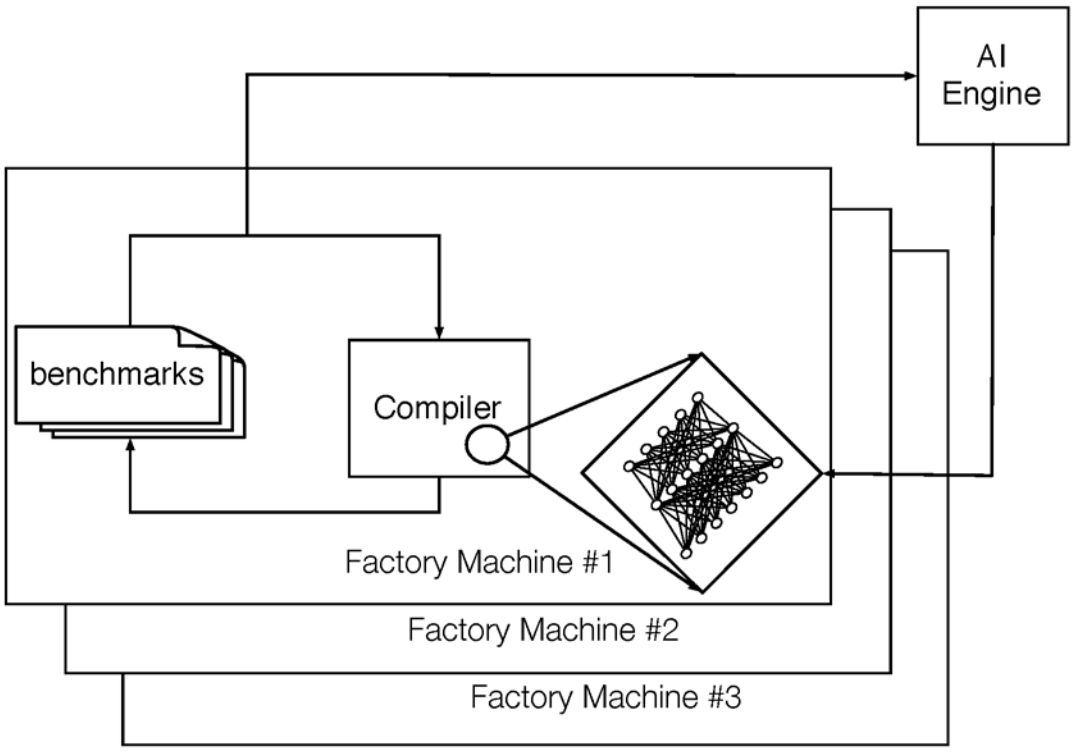


Figure 2.2: Deployment architecture

In the process of this research the training process is considered to be the time

spent on tuning the compiler. During this phase the AI engines present different ANNs, or GAs that need to be evaluated. The evaluation is done by compiling a given set of benchmarks using the AI to guide the optimization process. The compiled code is then run and the performance of the code is measured and compared to the baseline. This is a process that is trivial to parallelize and thus the training process can be distributed over multiple machines. Each machine in the cluster can independently compile and evaluate the ANN/GA provided by the AI engine and report back the result to the engine. This parallelization is shown in the Figure 2.2. The training phase is performed only once and once completed the compiler is ready to be shipped out to the consumers, the application developers in this case.

Modifications in the compiler

During the training phase, NEAT generates an ANN that is used to control the order of optimizations within Jikes RVM. The ANN is evaluated by applying different optimization orderings to each method within each training program and recording the performance of the optimized program. The ANN takes as input a characterization (called feature vector or source features) of current state of the method being optimized and outputs a set of probabilities corresponding to the benefit of applying each optimization. The optimization with the highest probability is applied to the method. After an optimization is applied, the feature vector of the method is updated and fed into the network for another round of optimization. One output of the network corresponds to "stop optimizing," and the optimization process continues until this output has the highest probability.

Deployment

Once the best Machine Learning algorithm is evolved, it is planted into the compiler that can make use of the decisions provided by the machine learning algorithm. During the execution phase the compiler would use the source code generator to generate source features that would be needed as inputs to the machine learning algorithm.

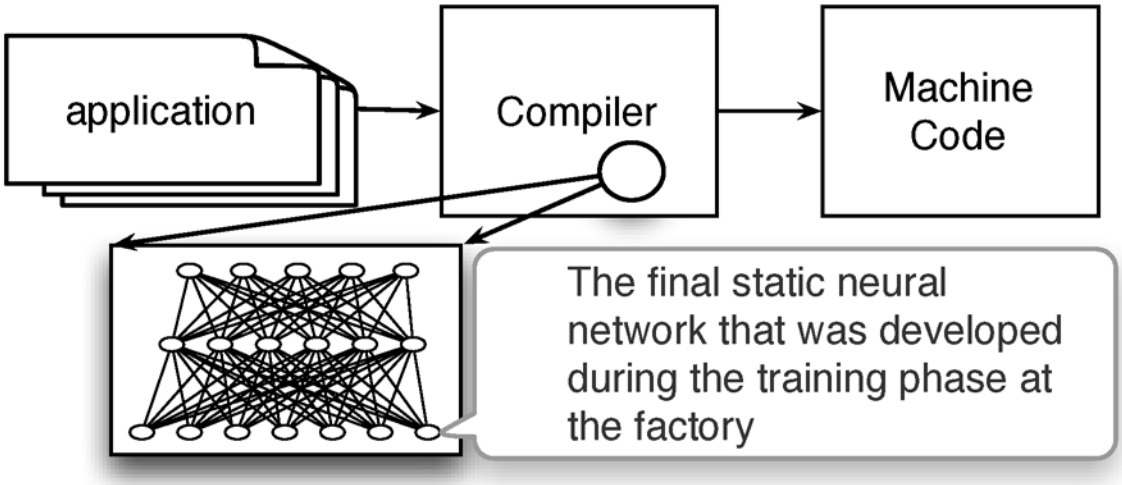


Figure 2.3: Deployment architecture

The source feature extractor has to be designed in such a way that only source features that require one single pass over the code are being collected, the limitation of a single pass is imposed to reduce the execution time and the time complexity induced by the modifications proposed by the methods in this research. Once the source features are collected, they are used as inputs to the ANN. The ANN is a weighted directed acyclic graph that has a set of input nodes, output nodes, and hidden nodes. The input nodes take in the input from the source feature extractor, and then the values are propagated over the different nodes in the network to collect the output values at the output nodes. These output nodes control the different optimizations available to the compiler and this helps us guide the optimization process in the compiler.

The ANN that is making these decisions would have been shipped from the *factory* along with the compiler, and would not change during the use at the application developer.

2.8 Neuro-Evolution Overview

This research primarily explorers use of Neuro-Evolution of Augmenting Topologies (NEAT) to construct our neural networks to be used for phase ordering. NEAT

uses a process of natural selection to construct an effective neural network to solve a particular task. This process starts by randomly generating an initial population (or generation) of neural networks and evaluating the performance of each network at solving the specific task at hand. The advantage of using an evolutionary approach to a static and traditional approach is the ability of the NEAT engine to produce ANNs that are smaller and less complex, another great advantage is in the ability of the NEAT engine to reduce the training time by multiple orders of magnitude. The Figure 3.2 shows the process of training in greater detail. The first generation of the ANNs that are generated by the engine are completely random. These are tested and their relative fitness is recorded. Only the best performing ANNs are propagated to the next generation, and used to generate the ANNs in the next generation. This process is repeated multiple times over many generations until we find an ANN that has the desirable fitness.

The number of neural networks present in each generation is set to 60 for our experiments. Each of these 60 neural networks is evaluated by using them to optimize the benchmarks in the training set. A fitness is associated with each network as described in Section 4.7. Once the initial set of generated neural networks are evaluated, ten best neural networks from this set are propagated to the next generation and are also used to produce new neural networks in the next generation.

This process continues and each successive generation of neural networks produces networks that performs better than the networks from the previous generation. New networks are created using mutation and crossover of the best networks from the previous generation. During the process of constructing new networks, we mutate the topology of a progenitor network. Mutation can involve adding a neuron to an existing edge in a network's hidden layer. We set the probability of adding a neuron to a low value (.1%) to keep our networks small and efficient. Mutation can also involve adding a new edge (probability .5%) or deleting an existing edge (probability .9%). These probabilities are within the ranges suggested by the authors of NEAT. Neurons

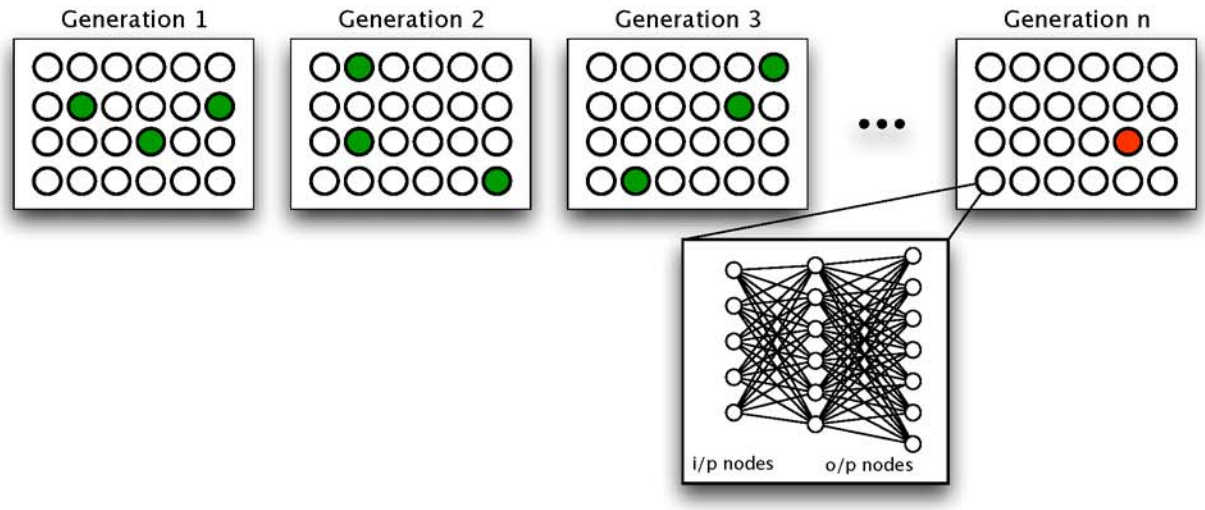


Figure 2.4: Training of the ANN using NEAT

are removed when the last edge to or from that neuron is removed. The mutation probabilities were manually-tuned for our specific task. For our present experiments, we stopped after 300 generations, which was when the performance of the networks no longer improved at the task of phase-ordering for our training benchmarks. Figure 3.2 depicts the process of constructing a neural network using NEAT to replace the optimization heuristic in Jikes RVM.

Applying NEAT

There are many characteristics (i.e., features) that can influence the phase-ordering decision, and these factors may have complex inter dependencies between them. In order to effectively model the non-linear behavior of these features, our neural networks are multilayer perceptrons.

Why NEAT?

In training traditional network, a fully connected ANN is provided by the developer trying to train the neural network. This ANN is then modified by tweaking the

weights of the connections to achieve a trained ANN. In this process the person generating the initial ANN would initially provide an overly simplistic ANN or an overly complex one. If the initial ANN is too simple in its structure, the ANN structurally be incapable of emulating the system that it is designed to emulate. If the initial structure of the ANN is more complex than required, it would exponentially increase the training time required to train the neural network.

NEAT overcomes this challenge as it evolves the networks of unbounded complexity from a minimal starting point. The NEAT engine can add and remove neurons as well as connections with the aim to keep the structure as small as possible. This method has been shown to outperform the best fixed-topology method on challenging reinforcement learning tasks [41]. The reasons that NEAT is faster and better than typical reinforcement learning can be summarized as:

1. it incrementally grows networks from a minimal structure
2. it protects structural innovation using natural selection
3. it employs a principled method of crossover of different topologies

Neural networks traditionally have been trained using supervised learning algorithms, which require a labeled training set. A labeled training set consists of a feature vector that is used as input, which characterizes a particular decision point and the correct label or desired output the network should produce when given this input. In our case of the phase ordering problem, we would need a feature vector corresponding to the code being optimized and the desired output would be the sequence of optimizations to apply to that code. Generating this labeled dataset requires knowing the right sequence of optimizations to apply to a method is difficult as discussed in Section 3.3.

Structure of the network

In our neural networks, each feature or characteristic of the method is fed to an input node, and the layers of the network can represent complex "nonlinear" interaction between the features. Each output node of the network controls a particular

optimization that could be applied. The outputs are numbered between 0 or 1 depending on whether the optimization is predicted to be beneficial to the state of the code currently being optimized. We apply the optimization pertaining to the output that is closest to 1 indicating the optimization that the network predicts will be most beneficial. One of the outputs of the ANN tells the optimizer to stop optimizing. When the probability of this output is highest, the optimizer stops applying optimizations to the method. Figure 2.5 represents the phase-ordering process. The process of phase ordering starts when the Jikes RVM optimizer receives a method to optimize. We iterate over the instructions of the method to generate the feature vector, and then provide these features to the neural network. The neural network then provides a set of outputs, which represent the probabilities of each optimization being beneficial. The optimization with the highest probability is applied to the code. Once the optimization is applied the code could potentially have changed. This would mean that the ANN would need to be consulted again, and a new target optimization is generated by the ANN. This optimization is then again applied tot the code being compiled. This process is done multiple time until the code reaches the lowermost part of Figure 2.5 where the optimization to be applied is the *stop optimization* optimization. When the probability of this output is highest, the optimizer stops applying optimizations to the method.

2.9 Decision Tree

Past work and motivation

During the course of this research we have taken an unusual step to convert the trained ANN into a decision tree. This choice of presentation has primarily guided by the ease of readability of the finding and the inner workings by a compiler writer or an application developer, and the ability to debug the compiler decisions if and when such a need arises. The place where this has been presented is during method inlining where we attempted to perform single optimization tuning. The problem at hand could not

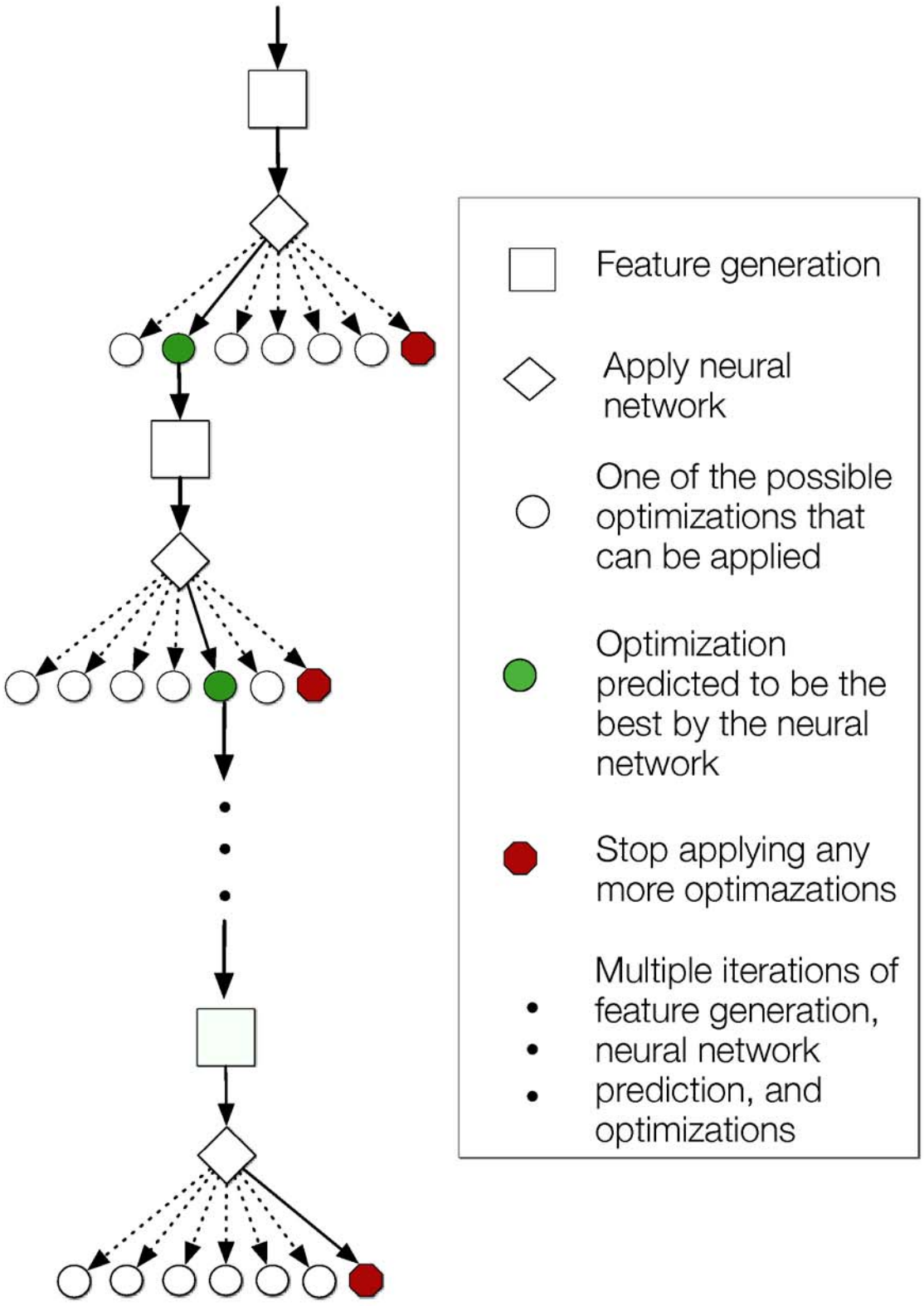


Figure 2.5: Phase ordering mechanism

generate a labeled dataset (explained in Section 4.7), and thus all normal methods of training decision trees were not available. This problem was solved by using a trained neural network to produce a pseudo labeled dataset. This dataset is then used to generate a decision tree.

Alexander *et al.*[42] presented the mathematical model to convert directed cyclic graph of a neural network with a time complexity of $O(n^2)$ for non-recursive graphs and $O(n^3)$ for recursive graphs. In our research we selectively limited our ANNs to non-recursive graphs, but the methods in this paper can be treated as the mathematical proof for algorithmic conversion of ANNs into Decision Trees.

There has been some research in converting ANNs into Decision trees in the past. Lu *et al.* [43] present a formal way to convert parts of a neural network into rules that can be combined into a decision tree. In this paper the authors de-construct the neural network and try to convert small portions of the graph that presents the neural network to *if-then-else* clauses mathematically and then combine multiple such *if-then-else* clauses to a decision tree.

Another approach that comes closest to our approach is is proposed by Craven *et al.*in [44] where they use the neural network as an oracle that can be used to a decision tree as an inductive learning problem. The difference between this work and ours is the way the tree is generated. The authors use a *best-first* approach to tree expansion as opposed to the conventional *depth-first* approach. Another difference between the methods used in this paper when compared to our method is in the stopping criteria. The leaf nodes are designed in a similar manner but the paper relies on limiting the total number of internal nodes as opposed to using the *max-depth* parameter that we used in our technique.

A similar approach is also presented in [45], where they use decision trees and ANNs to extract simple rules that can be used at a later time. This approach however deals with extracting symbolic rules from Decision Trees and ANNs, while we are trying to convert ANNs into Decision trees. This approach of using a Artificial Neural

Network as the oracle to generate a dataset can be used universally and has provided us with a very good approximation of more than 98.4% accuracy.

Creating training data

Artificial Neural Networks(ANNs) can be trained to understand complex tasks, however ANNs are difficult to understand for a human reader. Another disadvantage of using ANNs is the fact that they are by design black boxes and it is not possible to perform any debugging. In order to address this issue we attempted to convert the ANNs into decision trees. There has been some past research [44, 45, 43] in converting ANNs to decision trees that is described briefly in Section 2.9.

The traditional method of generating a decision tree is not available in our situation due to the absence of *labeled dataset*, explained in greater detail in Section 4.7. in order to get around this situation we just embed the ANN into the compiler and record each decision taken by the neural network. We also record the corresponding inputs that are provided to the compiler. Each of these input and output pairs is used as a single entry in the labeled training set for generating the decision tree. We used the standard C4.5 algorithm to generate the decision tree. Once the decision tree was generated we used static height tree pruning to reduce the max height. Doing this reduces the time complexity to *O(1)*, instead of being *O(h)* where h is the height of the tree. The final generated decision tree is shown in the Figure 4.10.

2.10 Fitness Function

The fitness value we used for the NEAT algorithm is the arithmetic mean of the performance of the benchmarks in the training set. That is, the fitness value for a particular performance metric is:

$$Fitness(S) = \frac{\sum_{s \in S} Speedup(s)}{|S|}$$

where S is the benchmarks in the training suite and Speedup(s) is the metric to minimize for a particular benchmark s, which in our case is the run time (i.e., running time of the benchmark without compile time).

$$Speedup(s) = Runtime(s_{def})/Runtime(s)$$

where s_{def} is a run of benchmark s using the default optimization order of optimization level O3. The goal of the learning process is to create a heuristic that determines the correct order of optimizations to apply to a particular method thereby reducing the running time of the suite of benchmarks in the training set.

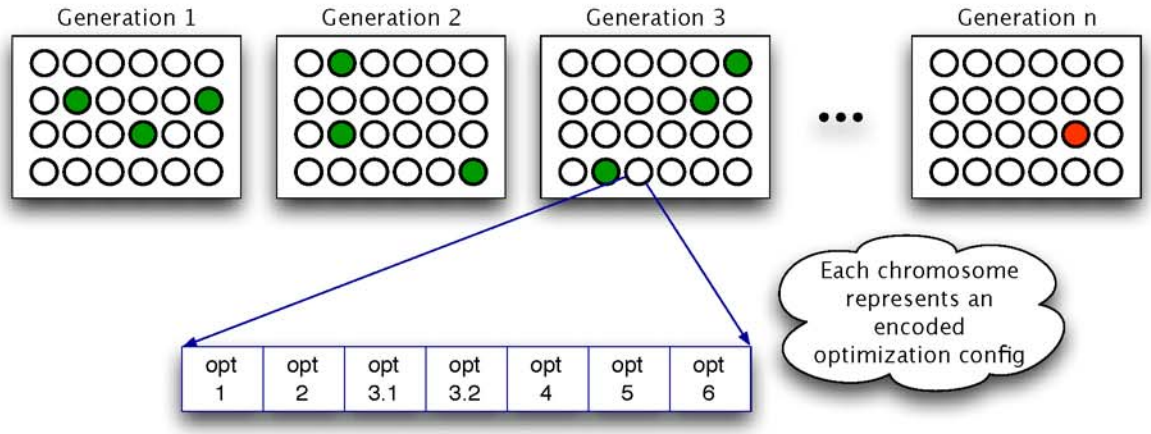


Figure 2.6: Training of the a chromosome using ECJ (GA)

2.11 Genetic Algorithms using ECJ

Genetic algorithm is a search method that uses evolution to perform fast search. GA works better than random search in being able to cover a larger search space in shorter amount of iterations and is better than hill climbing as it does not have a problem of getting stuck in local maximas. We use Genetic Algorithms to solve two different problems during this research. We used GA in machine learning to compare and contrast the difference in the effectiveness of using ANNs vs the use of Genetic algorithms. We also used Genetic Algorithms as the primary machine learning algorithm

to search for a good optimization configuration when we were tuning the performance of *FLib* when compiled using customized optimizations flags on GCC 4.8.

We used ECJ a Java-based Evolutionary Computation Research System, that provides a framework. This library like NEAT uses an evolutionary approach to finding and improving a specific heuristic. The Figure 2.6 shows the rough skeleton used during the training cycle. The ECJ generates a set of random chromosomes, and then tests out the fitness of each of the chromosomes. The fittest chromosomes are allowed to propagate and evolve the next generation. The goal of the ECJ engine would be to then mix two or more well performing chromosomes and create another chromosome that works better than all of the parents. This kind of evolutionary approach has proved to be demonstrably faster than exhaustive search or random sampling in complex search spaces.

Chapter 3

OPTIMIZATION ORDERING

Optimization ordering is the process of ordering a given set of optimizations in a specific order with the goal to increasing the performance of the code being compiled. This *Optimization Ordering* is also commonly called *Phase ordering* and we would use this phrase to reffer to Optimization ordering in this chapter.

During the compilation of a code from the original source code to machine code the compiler converts the source code to an intermediate representation. The compiler might work with multiple different intermediate representations(IRs) during the compilation process. During the compilation process the compiler applies different transformations to the IR. These transformations (or optimizations) take in the IR and modify the code and output code again in IR. This transformed code is again presented as input to another code optimization. This would mean that the two different optimizations would indirectly interact with each other as the output of one optimization is used an input for another optimization, thus the application of each optimization potentially affects the benefit of downstream optimizations.

There are different kind of optimizations, some are cleanup optimizations that are used regularly to clean up the code and perform basic housekeeping of the code after a specific optimization is applied, some might be enabling optimizations that would modify and transform the code in such a manner that another optimization might be able to work with the code in a more efficient manner. Another set of optimizations could be interdependent optimizations that might need to be applied multiple times in succession as the application of one optimization might create the need for the next optimization to be applied and vice a versa. It is easiest to visualize such optimizations

as perfect candidates for phase ordering, one of the most prominent examples of this is the phase-ordering problem between register allocation and instruction scheduling. However, any set of optimizations that interact with each other directly or indirectly are good candidates for phase ordering. These code interactions are an integral part of compiler optimizations, so it is important to understand the effects of the optimizations in order to arrange them in a way that can deliver the most benefit.

3.1 Phase-Ordering with Genetic Algorithms

Most compilers apply optimizations based on a fixed order that was determined to be best when the compiler was being developed and tuned. However, programs require a specific ordering of optimizations to obtain the best performance. To demonstrate our point, we use *genetic algorithms* (GAs), the current state-of-the-art in phase-ordering optimizations [29, 23, 4], to show that selecting the best ordering of optimizations has the potential to significantly improve the running time of dynamically-compiled programs.

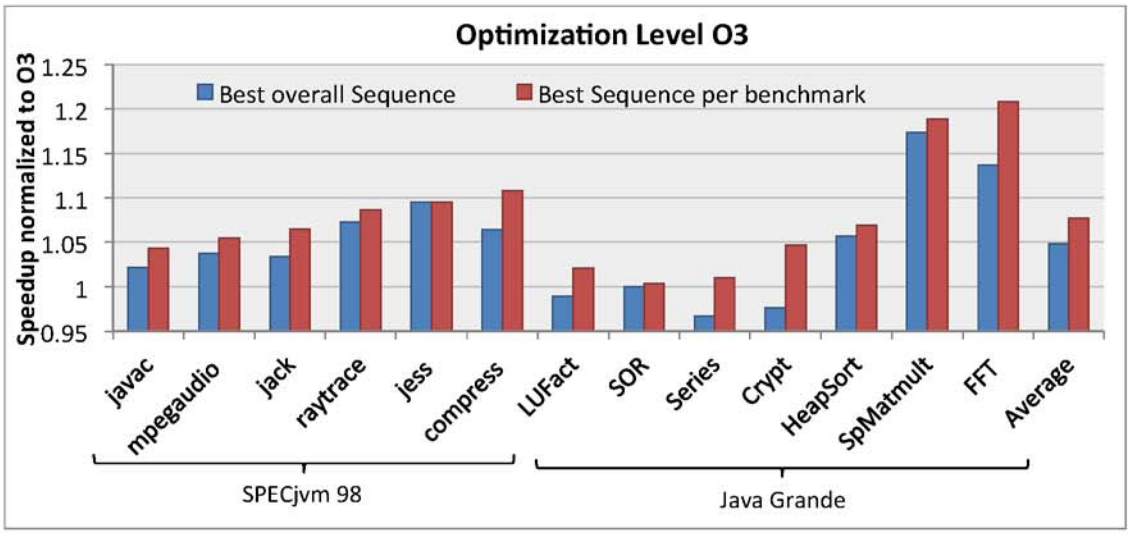


Figure 3.1: Performance of a generalized vs. customized sequence

Figure 3.1 shows a motivating example of using a compiler optimization sequence

specifically customized to a single application as compared to a single optimization sequence used for all possible applications. We used the Java Grande [46] and SPEC JVM 98 benchmarks for this example.[1] Genetic algorithm was used to chose the compiler optimization to be applied at each step for a given set of benchmarks acting as the training set. The "Best Overall Sequence" is generated by having all the benchmarks shown in Figure 3.1 in the training set, this created a optimization sequence that gave the best performance for all the benchmarks on average. The bar shown in the light grey color labeled "Best Sequence per benchmark" was obtained by performing a similar search space exploration on each of the benchmarks individually. This gave us an optimization sequence unique to each of the benchmarks being studied at that time. We see that this step provides a better performance by allowing the genetic algorithm to cater to each benchmark individually and thus increasing the performance of the complete set of benchmarks as a whole. This case can be logically extrapolated to searching for the best sequence of optimizations for each method in a particular benchmarks, but is a much harder problem we propose to address in this proposal. The results of these experiments confirm two hypotheses. First, significant performance improvements can be obtained by finding good optimization orders versus the well-engineered fixed order in Jikes RVM. The best order of optimizations per benchmark gave us up to a 20% speedup (`FFT`) and on average 8% speedup over optimization level O3. Second, as shown in previous work, each of our benchmarks requires a different optimization sequence to obtain the best performance. One ordering of optimizations for the entire set of programs achieves decent performance speedup compared to O3. However, the "Best Overall Sequence" degrades the performance of three benchmarks (`LUFact, Series`, and `Crypt`) compared to O3. Furthermore, searching for the best custom optimization sequence for each benchmark, "Best Sequence for Benchmark", allows us to outperform both O3 and the best overall sequence.

[1] We choose these benchmarks because they run for a short time. This allowed us to evaluate thousands of different optimization sequences using GAs.

Generating an optimization sequence using GAs: In our Genetic Algorithm, we create a population of strings (called chromosomes), where each chromosome corresponds to an optimization sequence. Each position (or gene) in the chromosome corresponds to a specific optimization from Table 3.2, and each optimization can appear multiple times in a chromosome. For each of the experiments below, we configured our GAs to create 50 chromosomes (i.e., 50 optimization sequences) per generation and to run for 20 generations.

Calculating the fitness function: We evaluate each optimization sequence (i.e., chromosome) by compiling all our benchmarks with each sequence. We recorded their execution times and calculated their speedup by normalizing their running times with the running time observed by compiling the benchmarks at the O3 level. We used average speedup of our benchmarks (normalized to opt level O3) as our fitness function for each chromosome. For the bars labeled "Best Overall Sequence" in Figure 3.1. For the second set of bars the fitness function for each chromosome was the speedup of that optimization sequence over O3 for one specific benchmark. This result corresponds to the "Best Sequence per Benchmark" bars in Figure 3.1. This represents the performance that we can get by customizing an optimization ordering for each benchmark individually.

3.2 Issues with Current State-of-the-Art

While the current state-of-the-art in phase-ordering of using genetic algorithms can bring significant performance improvements for some programs, this technique has several issues that impede its widespread adoption in traditional compilers.

Expensive Search

GAs and other search techniques are inherently expensive because they evaluate a variety (typically hundreds) of different optimization orders for each program and are therefore only applicable when compilation time is not an issue, e.g., in an iterative

compilation scenario. And, because there is typically no transfer of knowledge, the search space corresponding to the potential optimization orders has to be explored anew for each new benchmark or benchmark suite.

Method-specific difficulty

Using GAs to find a custom orderings of optimizations for code segments with a program (e.g., for each method) is non-trivial. An order of optimization specific to each piece of code requires a separate exploration of the optimization ordering space for that code. This requires obtaining fine-grained execution times for each piece of code after it is optimized with a specific phase-ordering. Fine-grained timers produce notoriously noisy information and can be difficult to implement. [2]

Note that exhaustive exploration to find the optimal order of optimizations is not practical. For example, if we consider 15 optimizations and an optimization sequence length of 20, the number of unique sequences exhaustive exploration would have to evaluate is enormous (15^{20}). Thus, the current state-of-the -art is to intelligently explore a small fraction of this space using genetic algorithms or some other search algorithm.

3.3 Proposed Solution

Instead of using expensive search techniques to solve the phase-ordering problem, we use propose to use a machine-learning based approach which automatically learns a good heuristic for phase-ordering. This approach incurs a one-time expensive training process, but is inexpensive to use when being applied to new programs. There are two potential techniques we could use to predict good optimization orders for code being optimized.

---

[2] Evaluating optimization orders for a method outside of an application context [47] can simplify fine-grained timing, but has the potential to identify optimization sequences that do not perform well when the method is used in its original context.

1. Predict the complete sequence: This technique requires a model to predict the complete sequence of optimizations that needs to be applied to the code just by looking at characteristics of the initial code to be optimized. This is a difficult learning task as the model would need to understand the complex interactions of each optimization in the sequence.

2. Predict the current best optimization: This method would use a model to predict the best single optimization (from a given set of optimizations) that should be applied based on the characteristics of code in its present state. Once an optimization is applied, we would reevaluate characteristics of the code and again predict the best optimization to apply given this new state of the code.

We focus on the second approach, which we believe is an easier learning problem to solve.

We used a technique called *Neuro-Evolution for Augmenting Topologies* to automatically construct a heuristic that can generate customized optimization orderings for each method in a program. The process of developing this heuristic is depicted in Figure 3.2 and described in detail in Section 2.8. The figure 3.2 represents the framework used to evolve and apply a neural network using NEAT to guide the compilation of a given method. The Figure 2.5 describes the way the neural network was used to guide the compilation process. This approach involves continually interrogating a neural network to predict which optimization would produce the best results as a method is being optimized. Our network uses as input features characterizing the current state of the code being optimized and correlates those features with the best optimization to use at particular point in the optimization process. As we are considering dynamic JIT compilation, the neural network and the feature generator must incur a small overhead, otherwise the cost of applying the network to perform phase-ordering might outweigh any benefits of the improved optimization orders.

Another approach would be to handcraft a heuristic based on experimentation and analysis. This is undesirable because it is an arduous task and specific to a compiler, the platform and the code being compiled. If any of these three parameter change, the entire tuning of the heuristic would have to be repeated.

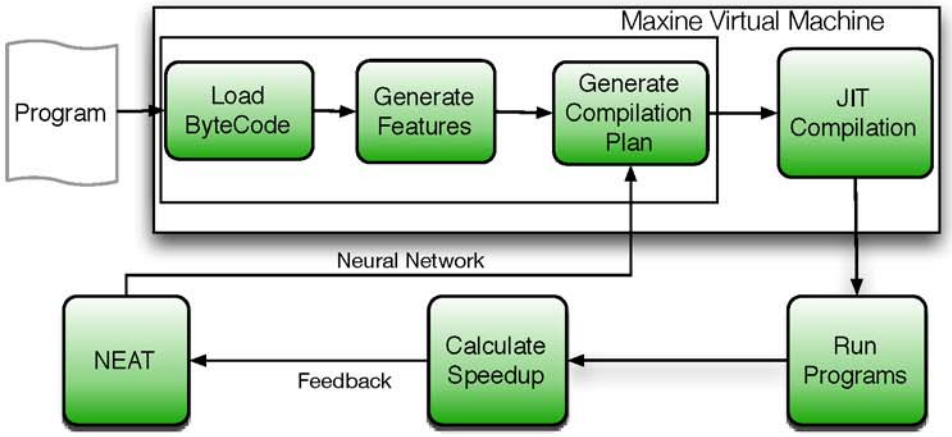


Figure 3.2: Block diagram of the compiler explaining Phase-ordering setup

3.4 Feature Extraction for Phase Ordering

The only way that we characterize the method being compiled is by generating static source code features. In this section we would talk about using static source code features, how we collect it, and other ways of characterizing the code.

Static source code features

Code features are usually a vector. Code features can be divided into two kinds static features and dynamic features. The primary difference between the two kind of features is the need to run the code in order to collect dynamic features. Static features are the set of features that can be collected by just parsing the code and there is no need to run the code in order to collect any run time characteristics. There are two primary reasons that we decided to use static source code features as opposed to dynamic features. Firstly using any dynamic features would limit any solution so found to only JIT compilers, or a very convoluted setup of combining a static compiler with past profiling runs. And secondly not using dynamic features would help avoid adding any more burden on the run time system.

Determining the properties of a method that predict an optimization improvement is a difficult task. As we are operating in a dynamic compilation environment,

1. NEAT constructs an ANN
   (a) Integrate the ANN into Jikes RVM's optimization driver
2. Evaluate ANN at the task of phase-ordering optimizations
   (a) For each method dynamically compiled, repeat the following two steps
      i. Generate a feature vector of current method's state
      ii. Use ANN to predict the best optimization to apply
3. Run benchmarks and obtain feedback for NEAT
   (a) Record execution time for each benchmark optimized using the ANN
   (b) Obtain speedup by normalizing each benchmark's running time to running time using default optimization heuristic (e.g., opt level O3)

Figure 3.3: Creating and evaluating ANN generated by NEAT

we chose features that are efficient to calculate and which we thought were relevant. Computing these features requires a single pass over the instructions of the method. Table 3.1 shows the 26 features used to describe the current state of each method being optimized. The values of each feature will be an entry in the 26−element feature vector x associated with each method. The first 2 entries are integer values defining the size of the code and data of the method. The next 6 are simple boolean properties (represented using 0 or 1) of the method. The remaining features are simply the percentage of byte codes belonging to a particular category (e.g., 30% loads, 22% floating point, 5% yield points, etc.).

3.5 Experimental Setup

In this chapter we describe the platform, the benchmarks, and the methodology employed in our experiments.

Feature	Meaning
byte codes	Number of byte codes in the method
locals space	Number of words allocated for locals
synch	Method is synchronized
exceptions	Method has exception handling code
leaf	Method is a leaf (contains no calls)
final	Method is declared final
private	Method is declared private
static	Method is declared static
Category	Fraction of byte codes that are...
aload, astore	Array Loads and Stores
primitive, long	Primitive or Long computations (e.g., iadd, fadd)
compare	Compares (e.g., lcmp, dcmpl)
branch	Branches (forward/backward/cond/uncond)
jsr	a JSR
switch	a SWITCH
put/get	a PUT or GET
invoke	an INVOKE
new	a NEW
arraylength	an ArrayLength
athrow,checkcast,monitor	are an Athrow, checkcast, or monitor
multi_newarray	are a Multi Newarray
simple, long, real	are a Simple,Long, or Real Conversions

Table 3.1: Source features collected during Phase ordering

Hardware and Operating System

Due to the large training time involved we used a cluster of Linux nodes to distribute the task among multiple machines. The total number of machines being used at any given time changed according to availability, but it ranged between 5 to 26. Each of the machine was a Intel Xeon X5680 CPUs running at 2.33 GHz with 8GBs of RAM.

Compiler

For our experiments in this research, we modified version 3.1.1 of the Jikes Research Virtual Machine [48]. The VM was run on an Intel x86 based machine,

supporting two AMD Opteron 2216 dual core processors running at 2.6GHz with an L1 and L2 cache and RAM of 128K, 1M and 8GB, respectively. The operating system on the machine was Linux, running kernel 2.6.32. We used the FastAdaptiveGenMS configuration of Jikes RVM, indicating that the core virtual machine was compiled by the optimizing compiler at the most aggressive optimization level and the generational mark-sweep garbage collector was used.

Benchmarks

For the present set of experiments we used four benchmark suites. For our training set, we used seven benchmarks from the Java Grande benchmark suite [49]. These benchmarks were used for training primarily due to their short execution times.

For the test set, we used the SPECjvm98 [50], the SPECjvm2008 [51], and the DaCapo benchmark [52] suites. We used all the benchmarks from SPECjvm98 and the subset of benchmarks from SPECjvm2008 and DaCapo that we could correctly compile with Jikes RVM. We used the largest inputs for all benchmarks. [3] The SPEC JVM benchmarks have been designed to measure the performance of the Java Runtime Environment (JRE) and focus on core Java functionality. The DaCapo benchmark suite is a collection of programs that were designed for various different Java performance studies. The results in Section 3.7 come from the benchmarks in our test set.

3.6 Optimization Levels

We ran our experiments in two scenarios, first using only the optimizing compiler in a non-adaptive scenario and second using the adaptive compilation mode. In the optimizing compilation scenario, we set the initial compiler to be the optimizing compiler and disable any recompilation. This forces the compiler to compile all the

[3] Note that for the benchmark FFT in SPECjvm2008, we used the small input size because the large input size required more memory than was available on our experimental platform.

OptKey	*Meaning*
Optimization Level O0	
CSE	Local common sub expression elimination
CNST	Local constant propagation
CPY	Local copy propagation
SA	CFG Structural Analysis
ET	Escape Transformations
FA	Field Analysis
BB	Basic block frequency estimation
Optimization Level O1	
BRO	Branch optimizations
TRE	Tail recursion elimination
SS	Basic block static splitting
SO	Simple optimizations like Type prop, Bounds check elim, dead-code elim, etc.
Optimization Level O2	
LN	Loop normalization
LU	Loop unrolling
CM	Coalesce Moves

Table 3.2: Optimizations (and abbreviations) used in present phase ordering experiments.

loaded methods at the highest optimization level. Under the adaptive scenario, all dynamically loaded methods are first compiled by the baseline compiler that converts byte codes straight to machine code without performing any optimizations. The resultant code is slow, but the compilation times are fast. The adaptive optimization system then uses online profiling to discover the subset of methods where a significant amount of the program's running time is being spent. These "hot" methods are then recompiled using the optimizing compiler. During this process these methods are first compiled at optimization level O0, but if they continue to be important they are recompiled at level O1, and finally at level O2 if warranted. Available optimizations are divided into different optimization levels based on their complexity and aggressiveness. When using the neural network in the adaptive scenario, we disabled the optimizations that belonged to a higher level than the present optimization level being used.

Measurement

In a dynamic compiler like Jikes RVM, there are two types of execution times that are of interest, total time and running time. The total time of a program is the time that the dynamic compiler takes to compile the code from byte codes to machine code, and then to actually run the machine code. The running time of a program is considered to be just the time taken to run the machine code after it has been compiled by the dynamic compiler during a previous invocation. For programs with short running times the total time is of interest, as the compilation process itself is the larger chunk of the execution time. However for programs that are likely to run for longer duration, e.g. programs that perform heavy computation or server programs that are initialized once and remain running for a longer period of time, it is important to highly optimize the machine code being generated. This is true even at the expense of potentially greater compile time, as the compilation time is likely to be overshadowed by the execution of the machine code that has been generated by the dynamic compiler. The time taken to execute the benchmark for the first invocation is taken as the total time. This time includes the time taken by the compiler to compile the byte codes into machine code and the running of the machine code itself. The running time is measured by running the benchmark over five iterations and taking the average of the last three execution times, this ensures that all the required methods and classes had been preloaded and compiled. To compare our performance we normalize our running times and total times with the default optimization setting. This default compilation scenario acts as our baseline, which is the average of twenty running times and twenty total times for each benchmark. The noise for all benchmarks was less that 1.2% and the average noise was 0.7%.

Evaluation Methodology

As is standard practice, we evolve our neural network over one suite of benchmarks, commonly referred to in the machine learning literature as the *training* set. We

then test the performance of our evolved neural network over another "unseen" suite of benchmarks, that we have not trained on, referred to as the *test* set.

Network Evaluator

The Neural network evaluator basically reads in the network generated by NEAT [53], and activates it with the inputs that are provided by the code feature generator. The inputs are then propagated through the edges connecting the neurons and the final output is generated in the form of an array. This array represents the relative probabilities of each of the optimizations controlled by the neural network. The output that is the largest is considered to be the best, and is applied to the code being compiled. Once compiled and executed the running time of the benchmark is collected and normalized by the baseline. This provides us with the speedup of the benchmark compared to the default compilation scenario. We use this speedup to provide feedback about the effectiveness of the neural network we had used.

Noise in measured running times

The benchmarks that we were using had inconsistent running times, this was a major problem since our complete approach relied on the accuracy of the running times. Our immediate solution was to run the benchmarks more number of times and then average the result. Doing this just smoothed out the problems but it also had the effect of hiding the effect (bad or good) of the phase ordering proposed by the neural network. Another approach was to try and use the minimum time of a benchmark run and consider it to be the most accurate time. We do this based on the assumption that the minimum time represents the best placement in the cache/heap of the hot methods of a benchmark, thus limiting the noise only to the cold or the non essential

methods,providing better results when compared to taking the average times.

$$
\begin{aligned}
noise &= \frac{stdev}{t_{avg}} * 100 \\
stdev &= \sqrt{\frac{\sum_{i=0}^{n} (t_{avg} - t_i)^2}{n}} \\
t_i &= time\ of\ i_{th}\ run \\
t_{avg} &= average\ time\ of\ i\ runs
\end{aligned}
\tag{3.1}
$$

Program	Training time (Days)
SPECjvm98	
javac	2.2
mpegaudio	0.8
jess	1.3
compress	1.1
raytrace	.9
jack	1.6
SPECjvm2008	
fft	10.4
lu	5
monte_carlo	8
SPECjvm2008 contd.	
sparse	6
sor	5.1
DaCapo	
avrora	7.3
luindex	3.1
lusearch	3.3
pmd	3.6
sunflow	3.1
xalan	5.6
Average	**3.9**
Total	**70**

Table 3.3: Average training time by GA for each benchmark individually.

Program	GA	NEAT
Java Grande	4.4	4.91
Jolden	7	8.3
Total	11.4	13.2

Table 3.4: Time taken in days to train the training set, to provide the results in Figure 3.7

Training Time

This section discusses the rough training time involved in the method inlining experiments that we conducted for this proposal. The reasons for us to use machine

learning is primarily because of the impracticality of exhaustive enumeration. Time taken for an experiment to complete played an important part in modeling the experiments, and this section gives a brief overview of the same. During training our machine learning heuristic requires us to provide fitness values to each of the heuristics being tested. This fitness value can only be generated by running the benchmark with the heuristic being tested and comparing the running time with the baseline.

This makes the execution time of the benchmark the bottleneck in our experiments. In order to give a clearer picture we calculated the rough training time that was required to train a phase-ordering sequence for each benchmark individually when using genetic algorithm. This is shown in the Table 3.3. Given the number of days that it can take to train each benchmark we feel that it would be impractical to use GA's for phase-ordering, especially within a dynamic compilation scenario.

3.7 Results

In performing the phase ordering experiments we trained a neural network on a set of training benchmarks viz. Java Grande and the jolden benchmarks and testing the performance of the neural network on a different set of benchmarks called the test set. The benchmarks in the test set were, SPECjvm98 and the DaCapo benchmarks. The JikesRVM compiler was invoked in two different modes, as an adaptive compiler and as an optimizing compiler.

Adaptive Compiler

The graph above represents the speedup achieved by using NEAT when used by Jikes RVM in adaptive mode to optimize each benchmark in the test set. We compare our result with the performance of each of the benchmarks when using the default adaptive compilation scenario.

In the adaptive compilation scenario, we allowed the adaptive compiler to decide the level of optimization to be used to optimize methods as described in Section 3.6.

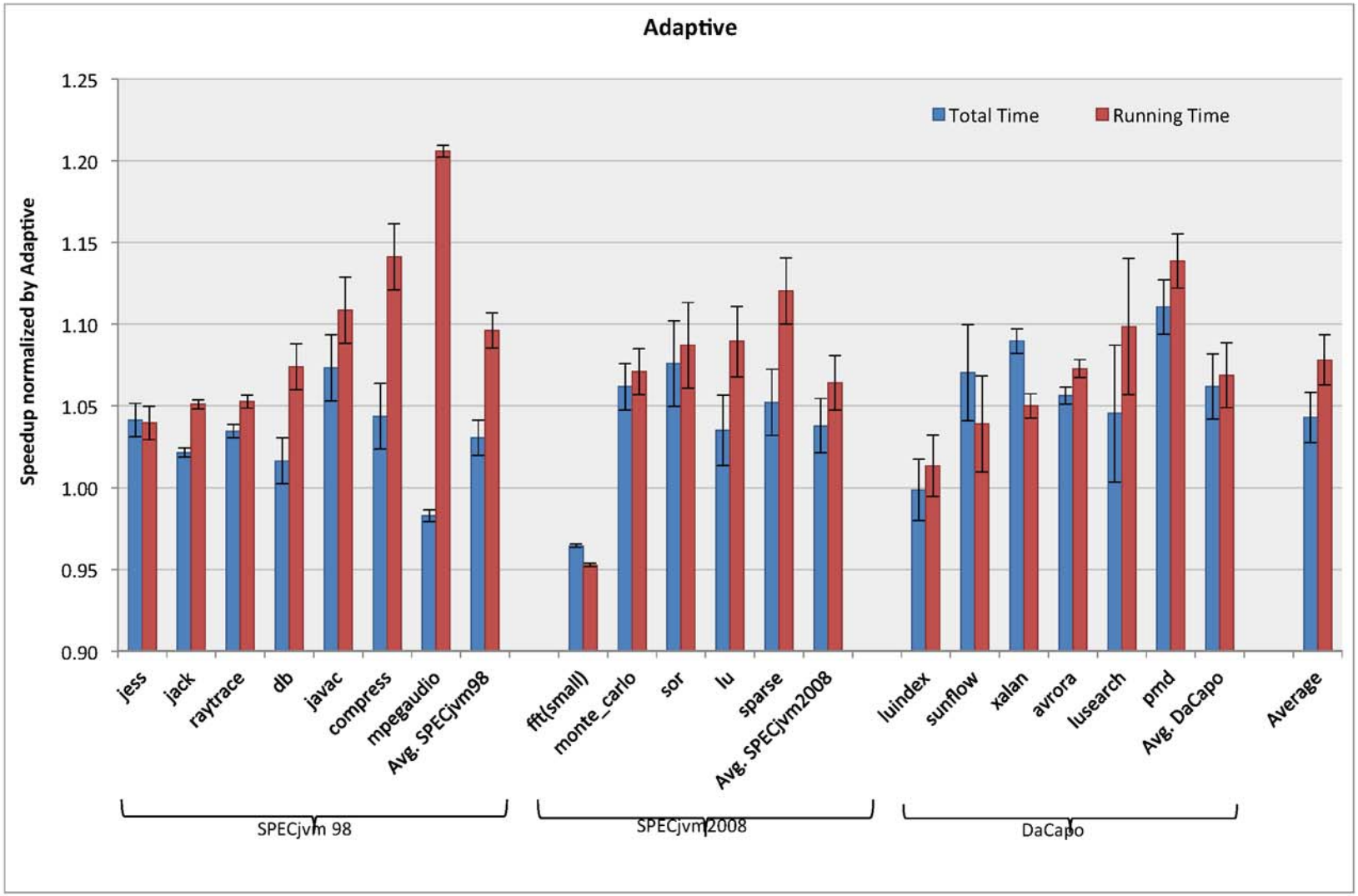


Figure 3.4: Performance of NEAT in Adaptive Compilation scenario.

However, at each optimization level we used the induced neural network to decide to order of optimizations to apply at that level. In this scenario, we obtained an average speedup of 8% in running time and 4% improvement in the total execution time over all the benchmarks versus the default adaptive mode in Jikes RVM.

SPECjvm98

Running time

Using our neural network for phase-ordering, we were able to obtain an average speedup of 10% across the seven benchmarks of the SPECjvm98 benchmark suite on the running time. We got significant improvements over default on mpegaudio (20%), compress (14%), and javac (11%).

Total time

We observed a modest increase in performance of 3% on average on the SPECjvm98 benchmarks. However, it is important to note that we achieved these speedups despite of the overhead of feature extraction and the execution of the neural network. The javac program gave us the best total time speedup at around 7%.

SPECjvm2008

Running Time

We achieved an average running time speedup of 6.4% on the SPECjvm2008 benchmarks. The `fft` benchmark did give us a slowdown of a little less than 5%. Interestingly, we discovered that the neural network used very short optimization sequences to optimize that benchmark. This helps to explain the improvement in the total time for this benchmark as described in the next section.

Total Time

Our average performance improvement over all five SPECjvm2008 benchmarks was around 4%. We achieved a performance improvement of up to 7% on the benchmark `sor` with our ANNs.

DaCapo

The running time performance improvement of the programs in the DaCapo benchmark suite (at 6.8%) was not as high as the other two benchmark suites, but their performance on the total time of 6% was much better than the average of the other two SPECjvm benchmark suites.

Optimizing Compiler

When running Jikes RVM in a non-adaptive mode, all the methods are compiled directly at the highest optimization level. The average speedup when just measuring running time was 8.2%, and we improved the total time by over 6%. In our experiments fft of the SPECjvm2008 suite did not perform as good, but after looking at performance in detail we realized that the network only applied 11 transformation on average to the methods in fft. The sequence length of the default optimizing compiler was 23 transformations. This could explain the speedup obtained by fft in Figure 3.5.

SPECjvm98

Running time

In SPECjvm98, we achieve up to a speedup of 24% on `mpegaudio`. On average, we improved the running time performance of this benchmark suite by 10%, which is a significant improvement.

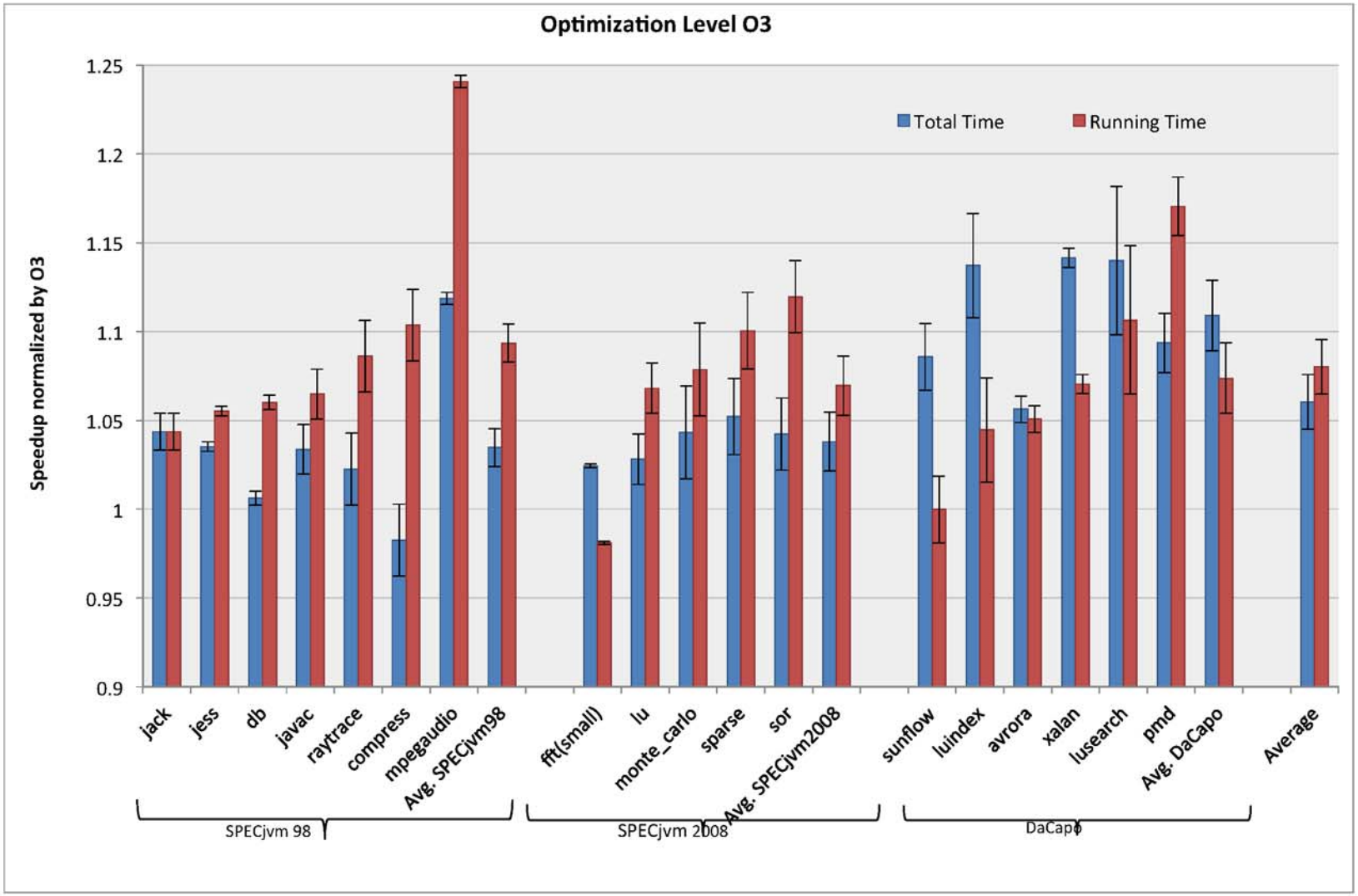


Figure 3.5: Performance of NEAT in Optimizing Compilation scenario.

Total time

When measuring total time, we observed a modest increase in performance of around 3.4%. The best performing benchmark was again `mpegaudio` at 11% speedup.

SPECjvm2008

Running Time

We achieved an average running time speedup of 7% over all the five benchmarks of the SPECjvm2008 benchmark suite. The best performing benchmark from the SPECjvm2008 suite was `sor` with a speedup of almost 12%.

Total Time

An interesting observation here is the performance of the `fft` benchmark. In all other cases this benchmark had a minor slowdown. We realized that the average optimization sequence length suggested by the neural network was 11. This is very short compared to the default fixed order sequence length of 23. This reduction in the sequence length helped to reduce the amount of compilation required, and thus improves total time performance.

DaCapo

Running Time

Using the Jikes RVM in a non-adaptive mode, we were able to get some significant speedups of 17% for `pmd` and 10.6% for `lusearch`. There were no significant slowdowns and on average we observed a speedup of 7.3% on the DaCapo benchmark suite.

Total Time

We saw significant speedups across DaCapo with 14% speedups on `xalan`, `luindex` and `lusearch`, and speedups of 5%, 8%, and 9% on the three other programs. On average, we had an improvement 11%.

3.8 Discussion

In this section, we briefly describe the neural network that we used for the experiments and discuss some observations (e.g., the reduction in the optimization sequence length, a case of repeated optimizations, and handling of relatively flat profiles.)

Neural Network

We used one neural network for all the results shown in Table 3.6 and Figures 3.4 and 3.5. This network had 30 inputs, 14 outputs, 24 hidden nodes, and 503 total connections.

Program	Avg. Seq. length
SPECjvm98	
javac	18
mpegaudio	19
jess	16
compress	19
raytrace	18
jack	17
SPECjvm2008	
fft	11
lu	18
monte_carlo	17
sparse	20
sor	16
DaCapo	
avrora	19
luindex	16
lusearch	16
pmd	18
sunflow	16
xalan	17
Average	**17**
Default	**23**
Reduction	**6**

Table 3.5: Average number of optimizations, applied by the trained ANN.

Benchmark	Hot method	Percent of Total Calls	Size	Optimization Sequence
SPECjvm 2008				
fft(small)	FFT.transform_internal()	86.93%	390	CNST,CPY,CPY,LU,BB,SS,BB,CSE,LN,CNST,LN
lu	LU.factor()	72.59%	277	TRE,CNST,CPY,SS,SS,BRO,SA,ET,SO, ET,LU,SS,LU,TRE,SS,SS,SO,CNST,FA,FA
monte_carlo	MonteCarlo.integrate()	25.31%	68	BB,CPY,BB,TRE,CNST,BB,CSE, CSE,LU,CSE,SS,SA,LU,FA
sparse	SparseCompRow. matmult()	80.79%	161	SO,BB,LU,CNST,TRE,LN,CPY,TRE,SS,CPY, SO,SO,SS,FA,BB,CNST,CPY,TRE,CNST
sor	SOR.execute()	86.51%	184	SO,SO,BB,SO,SS,CPY,ET,TRE,CPY,LN,CSE, CSE,SO,LN,SA,SA,SA,BB,TRE,CNST

Table 3.6: Best sequences for the hottest methods SPECjvm2008

Reduction of optimization sequence length

From our experiments, we were able to demonstrate two achievements. Intelligent ordering of the sequences provided us with significant speedups. We also show that intelligently applying the right optimizations helps in improving the compile time by not having to apply optimizations that have little impact on a method's performance. This would reduce the compilation burden on the system, and directly improve the system performance in terms of total execution time.

A detailed analysis of the phase orderings suggested by the ANN is shown in the Table 3.6. We typically applied 16-20 optimizations while the default optimizing compiler applied 23. We believe that this is significant. That is, we were able to apply the right optimizations and thus more effectively utilize the optimization resources available to us.

Repeating optimizations

In some cases the optimizations get repeated back to back. For example, the sequence shown in the fourth row of Table 3.6, the network predicted to apply *Static Splitting* twice in succession. This situation arises when applying a particular optimization does not change the feature vector. We could potentially be stuck in an infinite loop where the feature vector remains the same, thus inadvertently causing the neural network to apply the same optimization, which causes an infinite loop. In order to overcome this situation, if the network predicts that applying the same optimization again would be beneficial, we allow for a maximum of 5 such repetitions, and then instead apply the second best optimization.

Improvements from present state of art

At present the best way to tune phase ordering is to use GA to optimize in the search. There are a few problems with this approach, each benchmark has to be tuned individually, if we use a training set and a test set, the results are not as good

as shown in Figure 3.7. Figure 3.7 compares the present state of the part in phase ordering with our approach. The first bar is by training GA on a training set and testing it on the test set, similar to the second bar where we used NEAT. The last bar is when we individually searched for the best phase ordering using GA for each benchmark. Even with the advantage of being trained on each benchmark individually, the performance GA per benchmark is not much better than using NEAT, which does not require individual training runs.

Flat-profiled benchmarks

Most code written in normal applications and compiled by a compiler usually follow the *Pareto Principal* (also known as the 80-20 rule). The Pareto Principal states that as a thumb rule 80% of the execution time is spent in execution of 20% of the code base. This is an important inference in performing code optimizations, as the application developer can concentrate their efforts on optimizing just the important pieces of code and achieve good speedups with little effort. Some benchmarks however, the running time of the benchmark is equally divided among multiple methods (i.e., a flat profile), while other benchmarks have the majority of the execution time is spent in just one or a few methods. Finding a good phase ordering in case of benchmarks with one single "hot" method is relatively straight-forward. We would simply be searching for an optimization sequence that was beneficial for the one important method of the benchmark. We would just have to find a sequence that is beneficial for that one method and apply it to all the methods during that compilation instance, in our case it would be applied to all the methods that are loaded and or compiled during the execution of that benchmark. Since the execution time is dominated by a single method, we would see an overall improvement in the performance of the benchmark even if the method-specific phase ordering negatively affects the performance of the other methods.

For example, let us consider the example of the `lu` benchmark in the SPECjvm2008

benchmark suite. Looking at the profiling information, we realized that roughly 72% of the execution time was spent inside the measureLU() method. Now consider that a random search was to find a phase ordering sequence that improved the performance of this method by 10 percent, but this sequence also reduced the performance of all the other methods by 15%, we would still see a total improvement of 2.5-3%. So in cases where there is a dominance of a single method, we do not always have to worry about the impact of our phase ordering sequence on all the other methods. This method of finding the sequence that benefit just a few methods cannot be useful in all situations especially if work is evenly divided among multiple methods. for example if similar effects of the phases ordering sequence were true in the case of mpegaudio in the SPECjvm2008 benchmark suite, we would actually see a slowdown of almost 11%. In order to demonstrate our point, we conducted an experiment where we allowed the genetic algorithm to search for the best optimization sequence to be applied to each benchmark. The Figure 3.7 shows the comparison of using Genetic Algorithm to search for a single optimization sequence that would be applied to all the methods of a benchmark.

This approach of directed search of good optimization sequences for the hottest method was the method proposed by Cooper *etal.* [26] and was shown to find good optimization sequences for a program. Figure 3.6 shows the speedup achieved by both GAs and neural networks on each benchmark as it relates to the number of "hot" methods that constitute 60% of the running time for a particular benchmark. In this figure, we see that the GA is better at finding good speedups when the 60% of the execution time is concentrated in just one method. However, our NEAT-evolved networks are able to achieve good speedup when the execution time is distributed over multiple methods. Another set of results that reaffirm this conclusion is in Figure 3.7, if you look at the results for javac and mpegaudio, both benchmarks have relatively flat profiles, and in both cases the individually training GA phase ordering did not do as well as the Neural network.

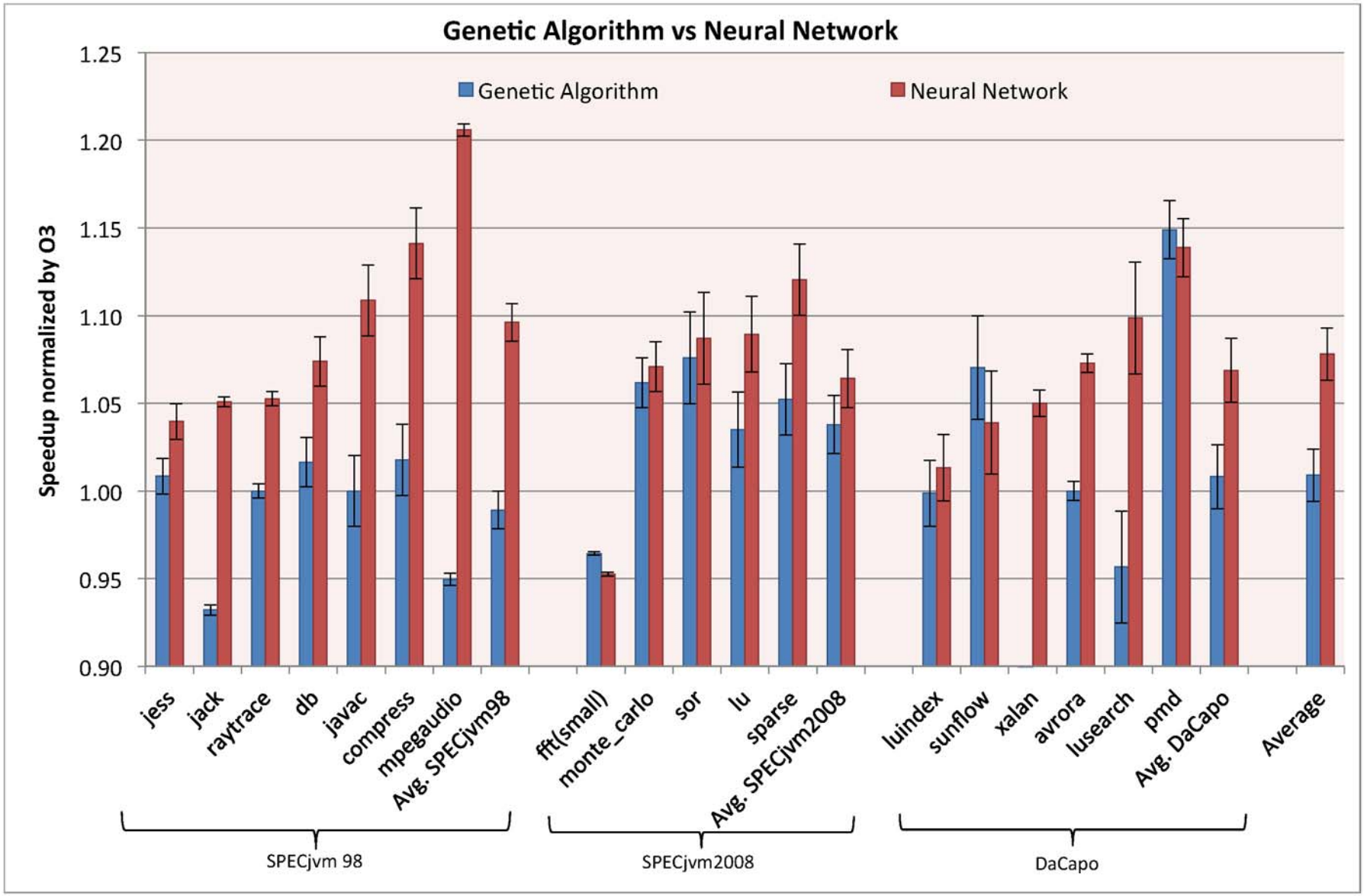


Table 3.7: Genetic Algorithm versus Neural Network

This particular property of the ANN to search for good optimization sequences even for relatively flat profiled benchmarks is even more important for newer architectures. Increasing number of cores would incentivize the application developer to divide the workload to multiple cores. The easiest way to achieve this would be to perform multiple different tasks on multiple cores, and in such situations we see that the solution based on NEAT would perform more reliably.

Exploration of Phase ordering benefit

In this section we try to analyze the optimization orderings that our neural network came up with. We ran the benchmarks and collected the profiling runs, which gave us an idea of which methods were most important. Looking at the neural network does not typically give any intuition of the phase-ordering heuristic, however it may help to understand the rough complexity of the final solution.

The first example that we look into is the `sparse.lu` benchmark in the SpecJVM2008 benchmark suite.

The neural network found interesting combinations of transformations that helped in improving the performance of some of the benchmarks. For example, the code shown in Figure 3.8 is the hottest method in the scimark.lu.small benchmark. The listing 3.1 shows code after applying *Branch Optimization* before *CFG Structural Analysis* (i.e., the ordering obtained from the default optimization level) and the code in listing 3.2 is obtained when applying these two optimizations in the reverse order (i.e., the ordering obtained from our neural network). We looked at the machine code being generated in both cases and realized that when *CFG Structural Analysis* was applied before *Branch Optimization*, the code that was generated had more branch statements. In the slower code, the loops are represented as "while loops", and the code that worked best had loops that are represented as do-while loops. This small difference in the machine code gave an improvement of approximately 8% in the running time of the `scimark.lu` benchmark. Because the original code had a large fraction of

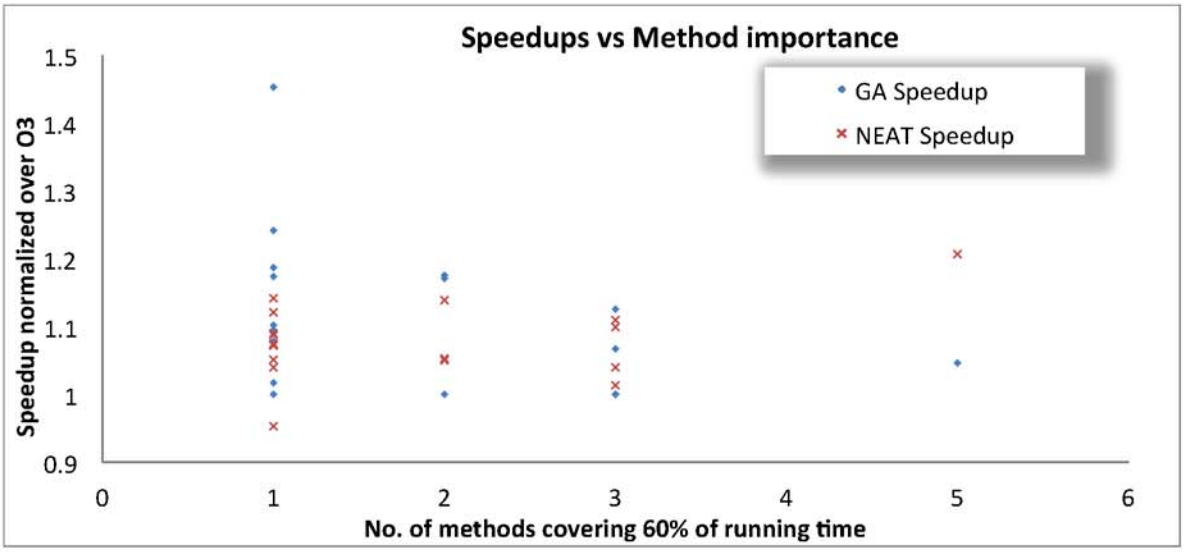


Figure 3.6: Speedup based on method importance

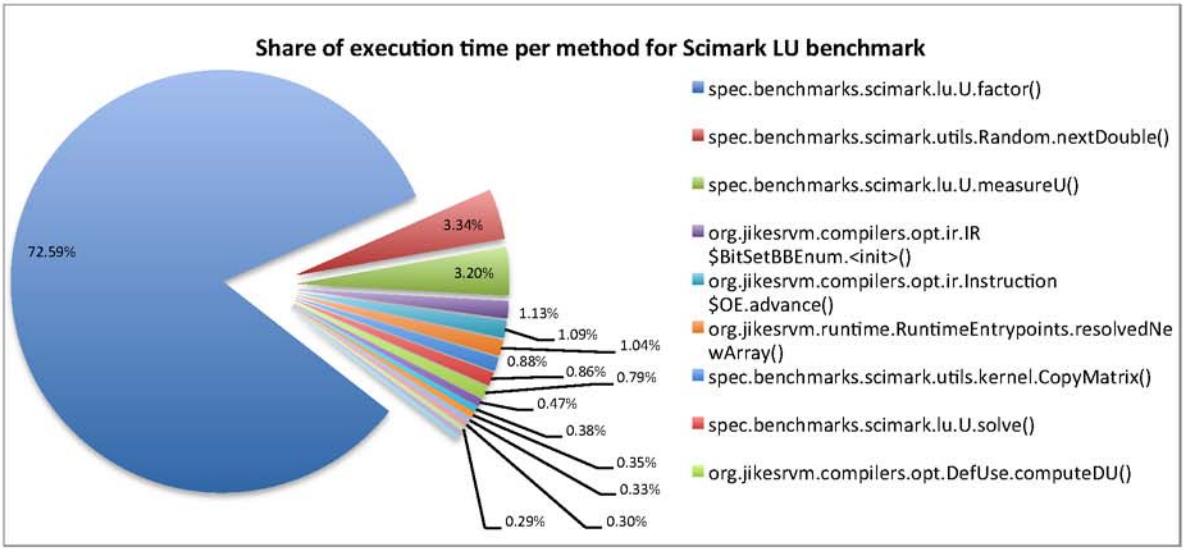
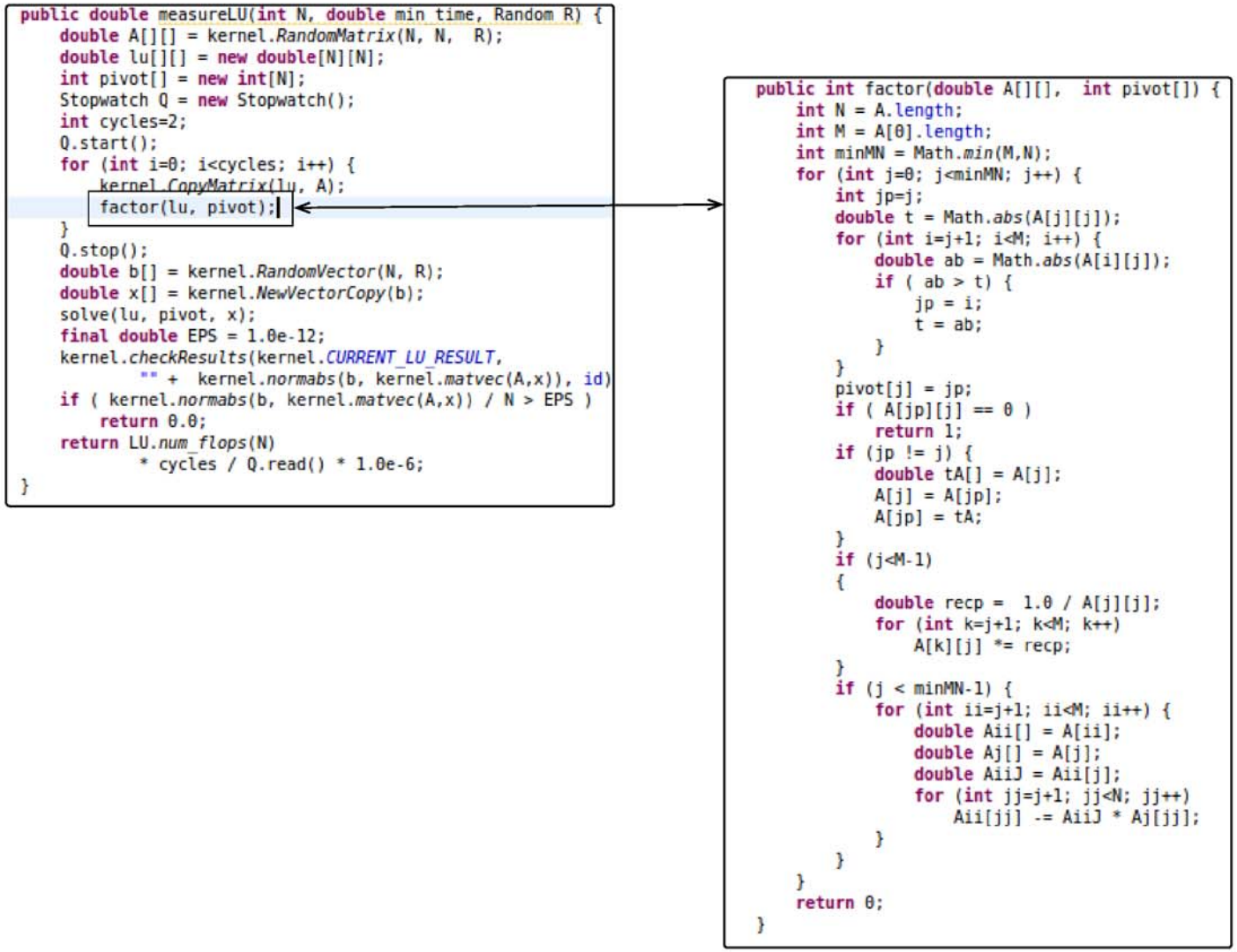


Figure 3.7: Speedup based on method importance



```
public double measureLU(int N, double min_time, Random R) {
    double A[][] = kernel.RandomMatrix(N, N,  R);
    double lu[][] = new double[N][N];
    int pivot[] = new int[N];
    Stopwatch Q = new Stopwatch();
    int cycles=2;
    Q.start();
    for (int i=0; i<cycles; i++) {
        kernel.CopyMatrix(lu, A);
        factor(lu, pivot);
    }
    Q.stop();
    double b[] = kernel.RandomVector(N, R);
    double x[] = kernel.NewVectorCopy(b);
    solve(lu, pivot, x);
    final double EPS = 1.0e-12;
    kernel.checkResults(kernel.CURRENT_LU_RESULT,
            "" +  kernel.normabs(b, kernel.matvec(A,x)), id)
    if ( kernel.normabs(b, kernel.matvec(A,x)) / N > EPS )
        return 0.0;
    return LU.num_flops(N)
            * cycles / Q.read() * 1.0e-6;
}
```

```
public int factor(double A[][],  int pivot[]) {
    int N = A.length;
    int M = A[0].length;
    int minMN = Math.min(M,N);
    for (int j=0; j<minMN; j++) {
        int jp=j;
        double t = Math.abs(A[j][j]);
        for (int i=j+1; i<M; i++) {
            double ab = Math.abs(A[i][j]);
            if ( ab > t) {
                jp = i;
                t = ab;
            }
        }
        pivot[j] = jp;
        if ( A[jp][j] == 0 )
            return 1;
        if (jp != j) {
            double tA[] = A[j];
            A[j] = A[jp];
            A[jp] = tA;
        }
        if (j<M-1)
        {
            double recp =  1.0 / A[j][j];
            for (int k=j+1; k<M; k++)
                A[k][j] *= recp;
        }
        if (j < minMN-1) {
            for (int ii=j+1; ii<M; ii++) {
                double Aii[] = A[ii];
                double Aj[] = A[j];
                double AiiJ = Aii[j];
                for (int jj=j+1; jj<N; jj++)
                    Aii[jj] -= AiiJ * Aj[jj];
            }
        }
    }
    return 0;
}
```

Figure 3.8: code for scimark.lu.LU.factor, the hottest method for the SpecJVM2008 lu benchmark

```
LABEL1
in_ifcmp <CONDITION> GOTO LABEL2
...
GOTO LABEL1
LABEL2
```

Listing 3.1: Slow Code (SA applied before BRO): generated by the default O3 compiler

```
in_ifcmp <!CONDITION> GOTO LABEL2
LABEL1
...
in_ifcmp <CONDITION> GOTO LABEL1
LABEL2
```

Listing 3.2: Fast Code (BRO applied before SA): suggested by our neural network.

Figure 3.9: Effect of optimization ordering in lu benchmark

unconditional branch statements, it triggered the neural network to apply *CFG Structural Analysis*. This kind of fine-grained optimization can be achieved when using a our method of phase ordering.

```
matmul()
{
  for (...) {
    arithmatic operation over an array
    for (...) {
      for (...) {
        arithmatic operation over an array
      }
    }
  }
  ...
  if(...){
    for (...) {
      arithmatic operation over an array
    }
  }
}
```

Figure 3.10: Pseudo-code for matmult, the hottest method for the SpecJVM2008 sparse benchmark.

Analyzing another benchmark, `scimark.sparse`, which performs sparse matrix multiplication, we see another similar phenomena. We looked at the `sparse.SparseCompRow.matmul` method, which is the hottest method in the benchmark and has multiple nested loops as represented in Table 3.8. Considering the

```
LABEL1
...
int_ifcmp
<CONDITION> GOTO LABEL3
goto LABEL2
LABEL2
goto LABEL4
LABEL3
goto LABEL1
LABEL4
```

Listing 3.3: Slow Code (LU applied before SA):generated by the default O3 compiler.

unrolling

```
LABEL1
...
int_ifcmp
<CONDITION> GOTO LABEL1
```

Listing 3.4: Fast Code (SA applied before LU): suggested by our neural network.

Figure 3.11: Change in machine code using different phase ordering

number of nested loops in this method, *Loop Unrolling* could potentially be an optimization to this method. However we realized that our neural network applied *CFG Structural Analysis* before it applied *Loop Unrolling*. This ordering helped in improving the quality of the code, improving the total running time by almost 14%. Again, this particular ordering is not present in the default ordering present in the JikesRVM compiler. There were some other differences in the machine code that were generated and the exact change in the machine that caused this huge speedup cannot be pinpointed, however we found a few instances of machine code that were less than optimal. Figure 3.3 shows a piece of machine code that is less than optimal. When looking at this particular instance we quickly realized that the code placement was needlessly complex. For example, if only the target of the first conditional jump was set to *LABEL1*, we would not need the last three unconditional jumps. Intuitively, a compiler writer would try to fix the problem by applying another optimization like *Branch Optimization* or applying *CFG Structural Analysis* once more. But, in this particular case repeating *CFG Structural Analysis* or applying another instance of Branch optimization did not improve the performance of the code, but instead applying static analysis before loop unrolling produced a better code as found by our neural network.

Chapter 4

OPTIMIZATION TUNING

Some compiler optimizations are more than just binary off/on operations. The non binary optimizations internally have one or more parameters that govern their behavior and how these optimizations effect the code that it is given to transform. Such parameters that can effect the performance of the code, and thus tuning these parameters can in turn affect the performance of the application being compiled. Optimization tuning is the process of tuning the parameters that effect the behavior of specific optimizations by the compiler in order to improve the performance of the code being compiled. We mention this in the Section 1.2, as one of the ways of performing compiler tuning. This chapter deals with tuning the compiler optimization "method inlining".

In the past there has been a lot of research done on improving method inlining. The list of parameters that have traditionally been used to tune method inlining can be divided into two streams, the first deals with controlling the aggressiveness of the method inlining that is applied to the code being compiled and the second set of parameters check the environmental factors that are present around the code that is being inlined. Most of the research that has been done on tuning method inlining has primarily focused on tuning the aggressiveness of the optimization itself and not necessarily on linking the aggressiveness to the environmental factors governing the code being optimized in itself.

Research presented in this chapter attempts to change this and presents a mechanism that characterizes the code being compiled and use this characterization as a means to guide the method inlining process. This chapter also gives a brief study on

different parameters that can be used and their relative importance in the method inlining decision making process. In order to build this relation between the code being compiled and the method inlining optimization, we use different machine learning algorithms and compare and contrast their relative advantages. We used two different machine learning techniques, Artificial Neural Networks by using the NEAT engine, and the Genetic algorithms. The theory and principals of the different machine learning algorithms themselves are presented in greater detail in Chapter 2. This chapter starts with a brief introduction to method inlining and it's importance, followed by the present solutions, our solutions and then compare the relative differences.

4.1 Introduction to Method Inlining

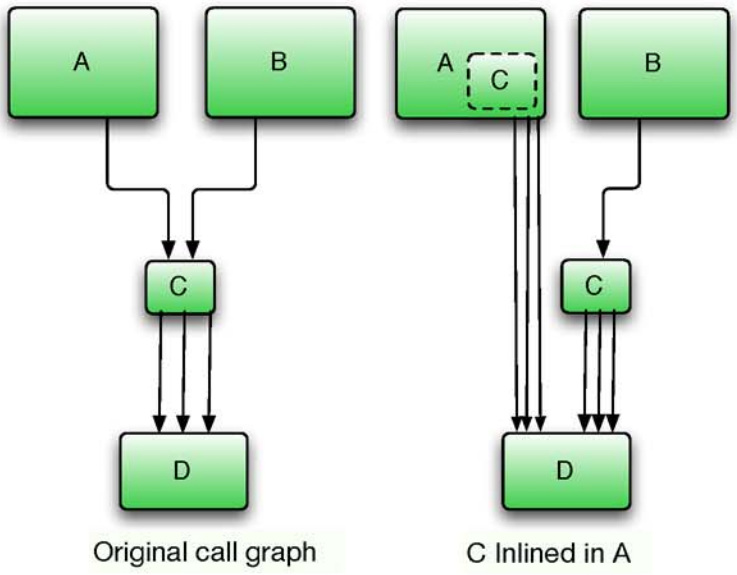


Figure 4.1: Diagram explaining an instance of Method Inlining

Method inlining is the process of embedding the body of one method (called the callee) at the instance of its invocation in another method (called the caller). The Figure 4.1 shows an example where method C is inlined into A, here the method C is the callee, and A is the caller. The figure 4.1 also shows that the callee is not inlined into the caller method B. This was represented to make it easy to understand that each

method inlining decision is unique and depends not only on the caller and the callee but also on the specific invocation.

Advantages of Inlining

If the method inliner decides to inline a specific method invocation, the invoke instruction is replaced by the actual code of the method being invoked. This internally means that the code used to save the execution context before invoking a new method and the piece of code that would load back the previously saved execution context once the execution of the new method ends is no longer needed, thus saving the context switching time. Another indirect benefit of inlining a method is that the compiler get to work on a longer piece of code, which might lead to other optimizations in the later stages of compilation. Most optimizations are only intra method optimizations being able to combine multiple methods would make it possible for the compiler to apply more aggressive optimizations to the code.

Disadvantages of Inlining

Aggressive inlining can also cause some performance degradation. When the compiler inlines aggressively, the body of the caller becomes large, this puts more pressure on the usage of the registers and leads to more memory spills. Another disadvantage of over ambitious method inlining is the increased memory footprint of the caller method, which also causes degradation in performance.

Method inlining is a balancing act, where too little inlining could leave performance improvements on the table underutilized, and too much inlining could cause sluggish performance due to memory spills or increased memory footprint. Getting the right balance is difficult but can provide a good boost to the performance of the application.

4.2 Importance of Method Inlining

Method inlining is one of the most important optimizations that effect the performance of an application, and has been a widely researched problem. In this chapter we would study and propose modifications to the method inliner on the Java HotSpot Server VM. The HotSpot VM is the most popular and widely used Java VM, any improvements on this production level server would present the most robust modifications towards method inlining.

Initial Study

There were two primary questions that we wanted to answer in order to understand the present state of the method inliner better.

1. The effectiveness of the present inliner
2. The density or sparseness of good method inlining configurations.

The execution time of SpecJVM benchmarks on the HotSpot Server VM with optimization level O3 was considered the baseline. We studied two different types of method inliners:

1. *No Inlining Inliner* This value was the relative performance of the benchmarks when the method inlining optimization was completely disabled.
2. *Random Inliner* In this set we used a random number generator to control the inlining decision.

We used the random inliner to run the set of benchmarks 1000 times and recorded the relative performance for each time, and the results are presented in the Figure 4.2. The red dotted line marks the performance achieved by completely disabling the inliner. This *No Inlining Inliner* achieved around 62% performance of the baseline. The blue line is the set of performances obtained by using the *Random Inliner* and then arranged in increasing order. What we see is that only two optimization configurations out of a thousand were as good as or better than the baseline method inliner, at the same time almost 10% of the configurations created by the random inliner were worse than actually not having any inliner at all. Based on this experiment there

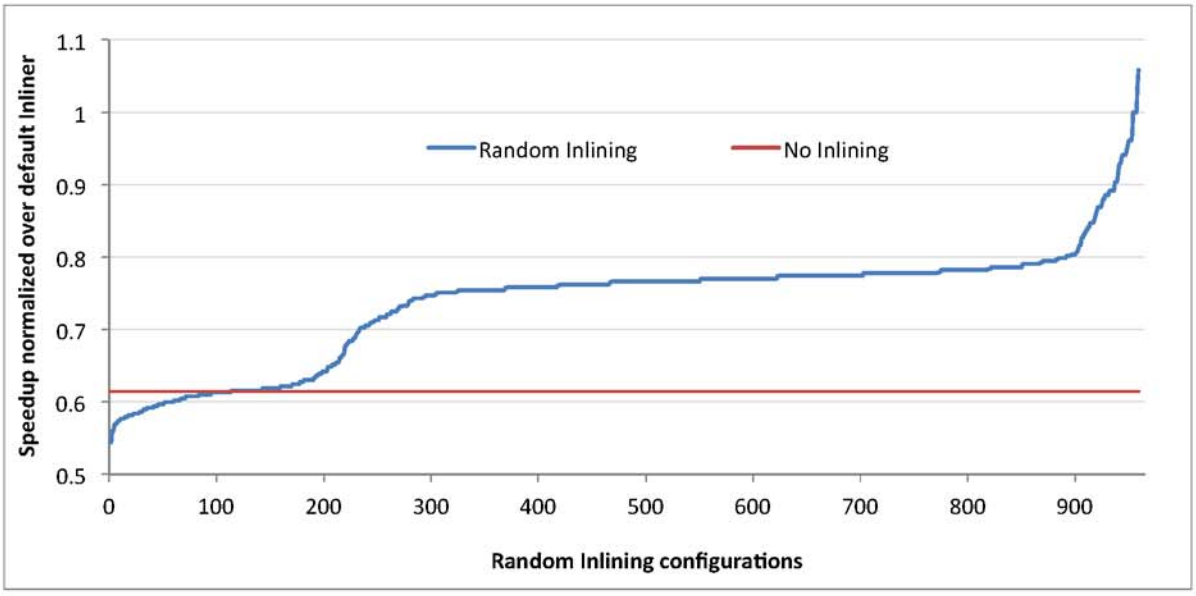


Figure 4.2: Performance of the random inliner on the raytrace benchmark

are two conclusions that can be made. One, the default inliner does an extremely good job at providing good method inlining configurations, and secondly method inlining if not applied intelligently could actually degrade the performance even more than the case of no method inlining at all.

4.3 Present Inlining Methodology

Most compilers at present use static rules to guide inlining, and these static rules are tuned based on large set of benchmarks. Figure 4.3 and Figure 4.4 show the pseudo code that is used by the two compilers to guide the method inlining optimization. There are a lot of similarities in the two inlining oracles, at the same time there are some differences as well that can be attributed to the ease of availability of certain information to the compiler. The compiler collects this information from the caller and the callee and uses it to make the inlining decision.

Figure 4.5 shows a high level representation of the default inlining oracle in the Java HotSpot Server VM, and we would discuss this in brief here as well.

Default Inlining Heuristic for two compilers

```
inliningHeuristic
(calleeSize, inlineDepth, callerSize){
    if (calleeSize > ALWAYS_INLINE_SIZE)
        if (calleeSize > CALLEE_MAX_SIZE)
          return NO;
    if (inlineDepth > MAX_INLINE_DEPTH)
        return NO;
    if (currentGraphSize >
      CALLER_MAX_GRAPH_SIZE)
        return NO;
    // Passed all tests so we inline
    return YES;
}
```

Figure 4.3: Inlining heuristic of the C1X compiler

```
inliningHeuristic
(calleeSize, inlineDepth, callerSize){
    if (calleeSize > ALWAYS_INLINE_SIZE)
        if (calleeSize > CALLER_MAX_SIZE)
          return NO;
    if (inlineDepth > MAX_INLINE_DEPTH)
        return NO;
    if (callWarmth >
      CALL_WARMTH_THRESHOLD)
        return NO;
    // Passed all tests so we inline
    return YES;
}
```

Figure 4.4: Inlining heuristic of the server compiler

Table 4.1 gives a brief introduction to the features that are being used by the two compilers. The algorithm used by default is simple to understand, and sets thresholds that keep the inliner in check. The method inliner looks at the callee method and the caller to make a customized decision about inlining, by collecting some information about each. Once this information is collected, the inliner uses the algorithms described in Figures 4.3 and 4.4, if the result is a yes, then the compiler would inline the callee method at the call site. For example if the callee size is less than trivial, which means that the code size of the callee is smaller than the context switching code, then method inlining would guarantee a reduction in the total code size as well as the elimination of the context switching time, such a case has all the advantages and none of the disadvantages and is a sure bet for inlining. However if the callee size is just a little larger than trivial size, the caller size is just about the max size, but the callee is extremely warm in such a situation the call on weather or not to inline is no longer as clear cut or simple to make.

4.4 Areas with Potential for Improvement

As shown in the Figure 4.2, the present inliners do a pretty good job at making a large set of the method inlining decisions. However there are some areas where

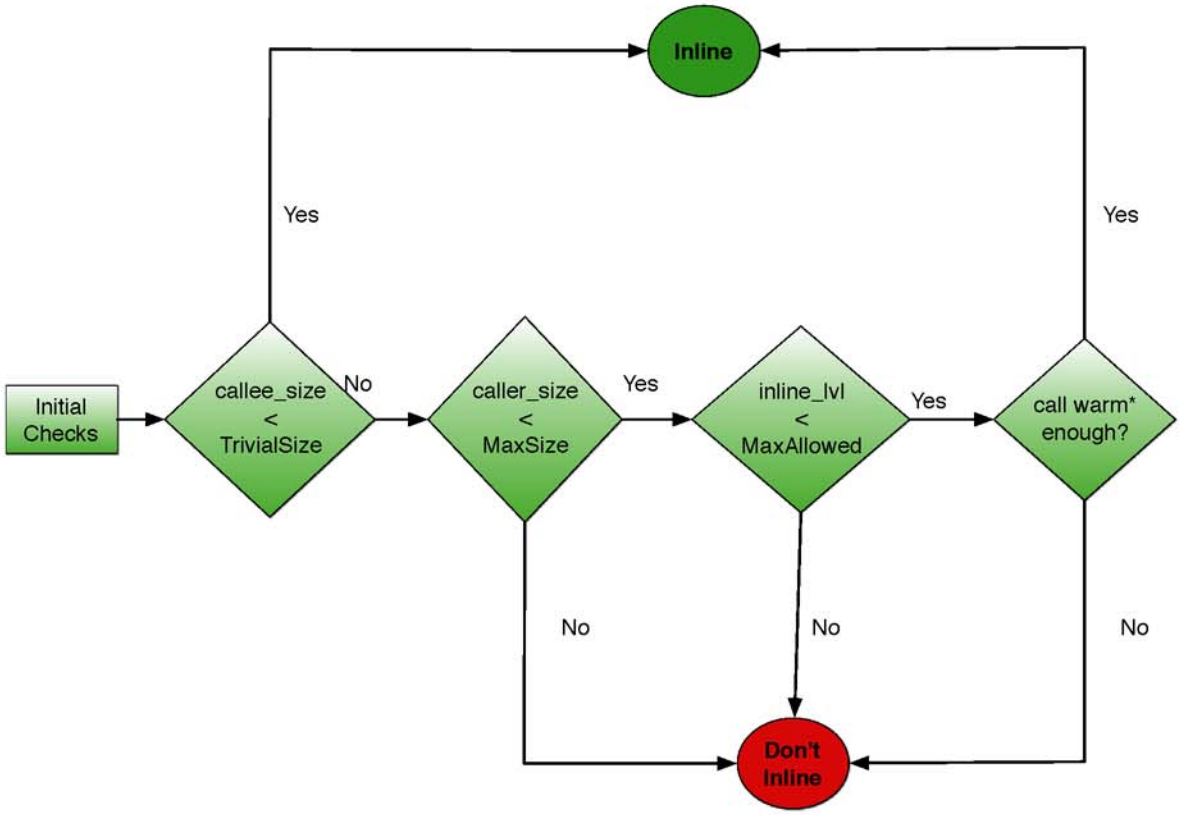


Figure 4.5: A high level representation of the default inliner on the Java HotSpot Server VM

the default method inliner could be improved by providing contextual information. Contextual information may or may not always be available. For the purpose of this study when using dynamic compilers a lot of this information is already available with the compiler or can be collected with little or no overhead. We briefly describe some of the information that could be collected and a small example where this would be useful.

Presence of Branch instructions

Methods that have a lot of branch instructions could potentially reduce the relative benefits of method inlining. Method calls that are nested deep in a multi level if-else loops might not be called as often as a method that is in the body of the code

Inlining Features	Description
ALWAYS_INLINE_SIZE	Callee methods less than this size are always inlined
CALLEE_MAX_SIZE	Maximum callee size allowable to inline
MAX_INLINE_DEPTH	Maximum inlining level at a particular call site
CALLER_MAX_GRAPH_SIZE	Max size of graph (number of IR nodes), which gives an estimate of the size of the root method plus methods already inlined
HotSpot Only Feature	
CALL_WARMTH	A compound heuristic that is a combination of call invocation counts, estimated profit from inlining and estimated amount of work done in the callee

Table 4.1: Features used by the default method inlining heuristic in Maxine VM and the HotSpot VM.

that is called every time. In such a situation inlining the method that is called every time would provide a better return on investment than inlining a method that might rarely be called. Another example could be method called during error handling.

Loops

Method inlining of a particular method call happens just once at each call site. If this call site is present in a loop that is going to be executed multiple times the application can take advantage of the performance benefits multiple times, and pay for the inlining overhead only once. Here inlining overhead would mean the computational cost of the actual inlining process and also the cost in terms of the increase in the memory footprint of the method.

Profiling based information

Dynamic compilers have another advantage of availability of profiling based information. During the initial code loading stage the methods are executed using the interpreter. During this phase the virtual machine can record the number of times each branch is taken. This information can provide a good platform for a branch predictor. In Figure 4.6 we show an example of using this branch prediction information to prioritize the basic blocks based on the probability of being called. During the

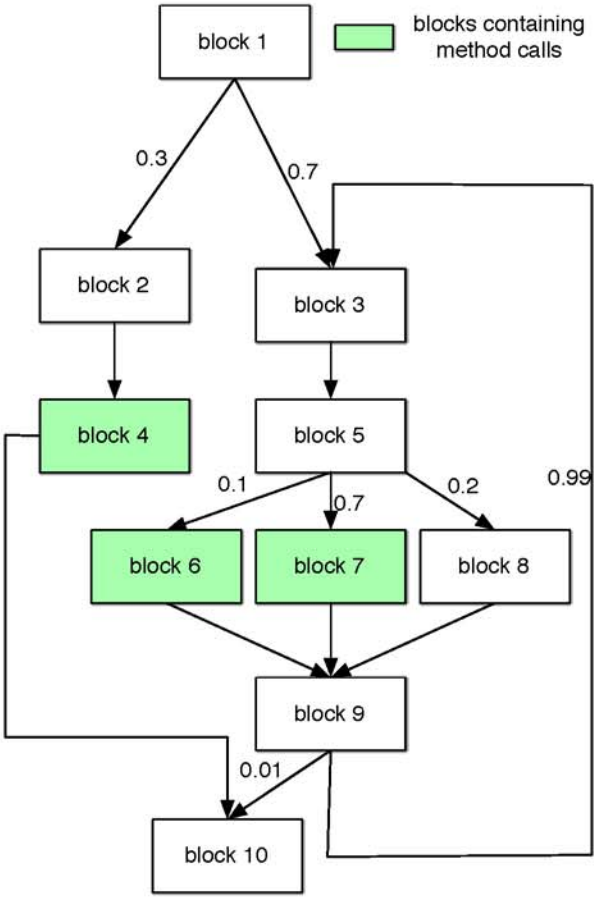


Figure 4.6: Calculating block weight

method inlining experiments this information turned out to be the most important parameter in performing method inlining.

Other method calls

If the caller method has a large number of methods being called, the importance of each of the callees reduce. For example if the caller method only has one method being called it would make sense that the compiler can inline that method without increasing the size of the code significantly. However if the caller method has multiple other methods that can potentially be inlined, it is very much possible that the method that is being inlined might not provide as much benefit as the method that we might encounter next.

Memory intensive or CPU intensive code

Memory intensive code does benefit with code motion or instruction scheduling that might happen in the future. Inlining methods that are memory intensive increases the chance that the compiler is able to hide memory latency better with instruction scheduling at a later point in the compilation process.

Other Source code features

Each of the parameters that we saw above can effect the method inlining decision. Also since each code is different there can be no single threshold that would work for all the code that is to be compiled by any compiler. In such a situation the compiler needs to make a decision at runtime to get the maximum possible performance. An example of just such a situation is shown in Figure 4.7. On the y-axis we plot the speedup normalized to the baseline, which in this case is the O3 optimization level with the default inliner. On the x axis we plot different max caller size values. The different lines on the graph represent different benchmarks from the SPECjvm benchmark suite. In this graph we see that there is no specific point at which the inlining oracle can create a threshold that would give the best performance over all the given benchmarks. This potential trade-off is inevitable if the method inliner uses one static limit or threshold for all possible permutations and combinations of code being compiled. However in case the method inliner is intelligent enough to understand the needs of the code based on characterization of the code it would be possible to have an optimal threshold set for each piece of code individually.

It is very likely that settings that are optimal for one set of applications might not be optimal for another set of applications. It might make sense to instead let the compiler decide which settings should be used based on the code being compiled and not some static set of rules that were fixed at the factory.

In order to be able to react favorably to these situations we collected the some information about the code being compiled just before the method inlining process.

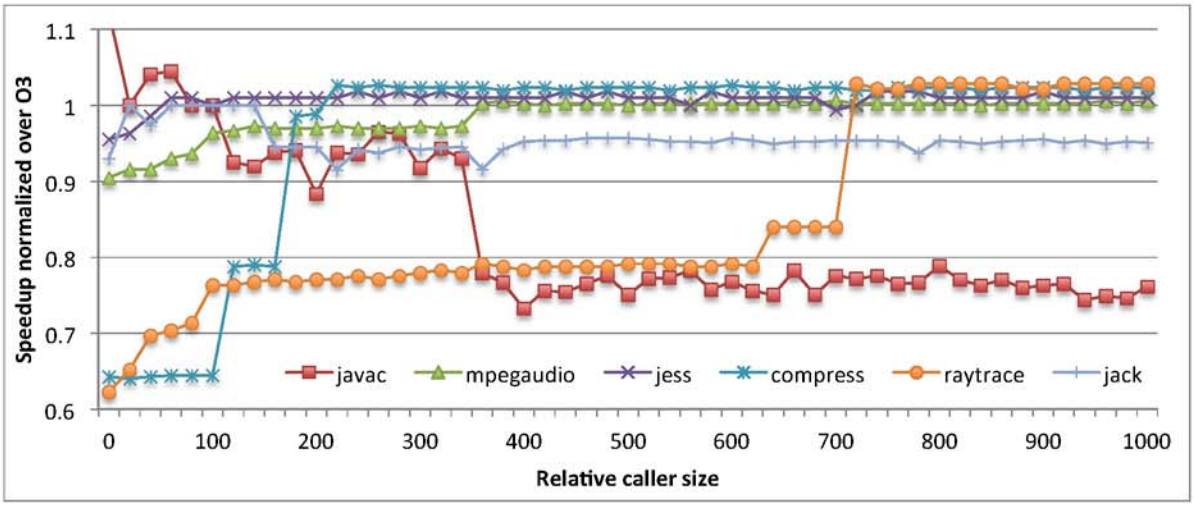


Figure 4.7: Performance of different benchmarks when using different caller sizes as thresholds.

The Table 4.2 above are source feature categories collected from an inlining decision point. We collect features corresponding to the instructions for the caller and the callee methods. We normalize these different instruction counts by total number of instructions and then use the caller and callee features as two different feature sets. We also show calling context features collected from the call site.

4.5 Other Proposed solutions

There have been many attempts in the past to improve method inlining. Some have been code specific and other have tried to be code agnostic. In this section we go through the present state of the art in method inlining methods. Cooper *et al.* [34] in their work present the use of Genetic algorithms to search for close to optimal method inlining thresholds. This work finds a good inlining points for benchmarks that are used for the study. However this work does not promise the same settings would be good for a new piece of code. Work by Arnold *et al.* [36] represents the inlining problem as a knapsack problem that calculates the size/speed trade offs to make inlining decisions. This work also does not present any promises for code that was not seen before. This is the drawback that this work and a lot of other past research have in common. Our

Feature	Description
Caller and Callee Features	
`Simple Instr.`	A static count of instructions that are typically CPU bound and do not require a memory access,
`Method Call Instr.`	Method calls (invoke instructions)
`Cond. Br. Instr`	Conditional branch instructions in a method
`Uncond. Br. Instr`	Unconditional branch statements in a method
`Memory Op. Instr.`	Load or store instructions
`New Obj. Instr.`	Instructions that create a new objects
`Default Instr.`	Instructions that did not fit in any of the above categories
`Size`	Total number of instructions.
Calling Context Features	
`InLoop`	Is the current call site in a loop?
`R-InlineDepth`	Inlining depth of a recursive call
`InlineDepth`	Inlining depth of a non-recursive call
`currentGraphSize`	Number of HIR nodes. Gives an estimate of caller method, including all methods already inlined
`Synchronized`	Is the called method synchronized?
HotSpot Only Features	
`Loop Depth`	if the block is a part of a loop, how deep is the loop.
`Block Weight`	Probability of the execution reaching the block

Table 4.2: Source Features collected during Method Inlining

approach creates a better heuristic that provides speedups for new code (benchmarks in the test set) as well as the benchmarks in the training set.

4.6 Search Space of Method Inlining Settings

The search space of potential good method inlining point can become extremely large even for really small applications. In this section we present rough calculations to explain this situation in more detail. Consider a sample very application *dummy-application* that has a total of 1000 methods. assuming that each method depends on at least calling 2 other methods, and there are no methods that are dead code, the total number of method calls is just 2000. This is a fairly simple situation, however the search space increases a great deal when you take into account that each method inlining decision changes the code that is being compiled. This leads the structure of the code to change itself, due to this mutation of code each inlining decision would implicitly effect all future inlining decisions that the inliner would need to make. The

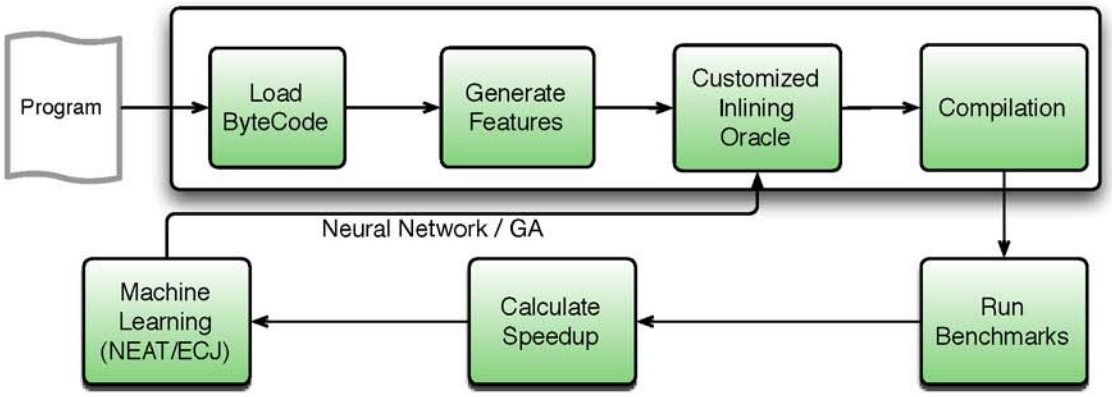


Figure 4.8: Framework used to construct effective inlining heuristics with machine learning

could lead to potentially 2^{2000} unique method inlining decisions. This is a huge number and makes it impossible to exhaustively enumerate and evaluate each possible final code. In the past research there have been some efforts to trim down this possible search space by using machine learning, however trying to approach this problem purely as a search space exploration can only provide localized solutions to specific problem at hand. The results obtained by expensive computational search of the possible search space would not be beneficial to any new code that is encountered by the compiler.

4.7 Approach

Based on the possible extensions and improvements that are possible we modified our compiler to take advantage of source feature analysis as well as profiling based feedback. The Figure 4.8 represents the framework that can effectively incorporate all the changes discussed in the previous section. The collection of these features and inputs can help make an informed decision, however the decision itself can be improved and tweaked. This is achieved using different machine learning techniques that we talk about in this section

Modifications to the compiler

The compiler being used was modified in two places.

1. *Source Feature Extraction* The source feature extractor was added just before the inlining decision needs to be made, as shown in the Figure 4.8 labeled by the block named "Generate Features". During the source feature extraction the feature extractor does one pass over the code of the two methods in question, profiling information is also collected during this stage.

2. *Method inlining Oracle* The method inlining heuristic that is generated by the machine learning algorithm is used as the inlining oracle. This heuristic needs to make all the inlining related decisions, and added into the compiler.

Training Infrastructure

The inlining decisions are taken by the inlining heuristic, all the parameters that are collected can be provided to the heuristic. There are many characteristics (i.e., features) that can influence the inlining decision, and these factors may have complex interdependencies between them.The number of parameters that effect method inlining is too large for a human being to construct a good inliner. To solve this problem we need to use machine learning to automatically generate a good inlining heuristic. We used two different machine learning algorithms (NEAT and ECJ) and compare the effectiveness of each of the techniques.

The NEAT and ECJ are both evolutionary approaches to generating Artificial Neural Networks(ANN) and Genetic algorithm respectively for the length of this section we refer to both of them as the heuristic. To training carried out to generate both are similar. The engines use a process of natural selection to construct an effective neural network to solve a particular task. This process starts by randomly generating an initial population (or generation) of neural networks and evaluating the performance of each network at solving the specific task at hand. The number of neural networks present in each generation is set to 60 for our experiments. Each of these 60 neural networks is evaluated by using them to compile and execute the benchmarks in the training set. Once all the randomly generated neural networks are evaluated, we use the best neural networks from this initial set (more than one may be chosen) to produce the next generation of neural networks. This process continues and each successive generation

of neural networks produces a network that performs better than the networks from the previous generation. The neural network uses as inputs the features described in Table 4.2 and produces an output indicating if inlining should be performed for a given call site. This process is repeated for a set number of generations and at the end of this evolutionary process we get a single heuristic that performs best. This heuristic is then used as the final heuristic to be used as the inliner.

Structure of the ANN

In order to effectively model the nonlinear behavior of features that effect the inlining process, the neural networks are modeled as multi-layer perceptrons. Each feature or characteristic of an inlining decision is fed to an input node, and the layers of the network can represent complex "nonlinear" interaction between the features. The output node of the network produces a number between 0 or 1 to depending on the decision that should be made at the particular inlining decision. If the output is greater than 0.5 we inline the particular call, else we don't inline. We use an unsupervised machine learning algorithm called Neuro-Evolution of Augmenting Topologies (NEAT) [54, 55] to construct an effective neural network to be used as an inlining heuristic. Figure 4.8 depicts the process of constructing a neural network using NEAT to replace the inlining heuristic in the Java Virtual Machines.

It is important to note that training and tuning a heuristic with machine learning happens *off-line*. Once the tuned heuristic is constructed, we can replace the default inlining heuristic with this new heuristic. Also, the final heuristic constructed by our machine learning approaches are as fast as any manually-constructed heuristics. Thus, there is no overhead incurred by using heuristics constructed with machine learning, nor are there any major code changes.

We show in Section 4.10 that NEAT can construct neural networks that are effective at solving the inlining decision problem. However, the induced NEAT heuristics (i.e., neural networks) are often unreadable which does not give the compiler writer

much insight or confidence in their utility.

Decision trees

There is one disadvantage of using ANNs. The ANN is basically a black box to a human reader, it is not possible to infer any information from looking at the network itself, this makes it very difficult to perform any debugging or manual fine tuning extremely difficult. However, there are other machine learning techniques, such as decision trees, that can generate learned models that are easy to read and understand. A decision tree can be described as an n-ary tree where each internal node represents a single feature or a collection of features that can effectively distinguish between a positive or negative instance. The leaf nodes represent final decisions provided by the tree. Decision trees are typical constructed using supervised learning methods. Supervised learning methods require a *labeled training set*, however, this *labeled training set* does not exist for the inlining decisions. A labeled training set consists of a set of input features characterizing a specific decision point and a labeled output that describes the correct decision ("label") to make for that particular decision point. For the problem of inlining, it is difficult to construct a labeled training set because knowing the correct output for a inlining decision point cannot be determined in isolation.One must consider an inlining decision point in the context of many other decision points. We use NEAT as discussed in the previous paragraph to overcome this difficulty. We thus use the heuristics generated by NEAT as a proxy for an oracle, because the heuristics generated by NEAT perform well at the task of deciding when and when not to inline.We discuss the creation of labeled training sets using NEAT heuristics in greater detail in Section 4.10.

Using NEAT networks to Construct Decision Tree Training Data

A labeled training set consists of input data representing a set of features that can characterize the decision point. We want to use the same set of features that are

used by NEAT to construct a readable decision tree heuristic. Along with the input features, an important component of the train set are the labeled outputs. These labeled outputs represent the decision that is thought to be optimal for a particular inlining decision.

We use the best neural network constructed by NEAT to predict the right label for a particular inlining decision. We give the neural network the input features corresponding to a single decision point, and record the network's output. This output is then used as the final label for that particular decision point. The tuple of the input features and output labels are then used as our training set to create a decision tree. Once this training set is created, we use the Weka toolkit [56] to generate a decision tree. This decision tree replaces the inlining heuristic in Maxine and is human-readable as well as easy to understand.

Fitness Functions

The fitness value we used for the NEAT and GA algorithms is the arithmetic mean of the performance of the benchmarks in the training set. That is, the fitness value for a particular performance metric is:

$$Fitness(S) = \frac{\sum_{s \in S} Speedup(s)}{|S|}$$

where S is the benchmarks in the training suite and Speedup(s) is the metric to maximize for a particular benchmark s.

$$Speedup(s) = Runtime(s_{def})/Runtime(s)$$

where s_{def} is a run of benchmark s using the default heuristic. The goal of the learning process is to create an inlining heuristic that reduces the running time of the suite of benchmarks in the training set.

4.8 Experimental Setup

In this section we talk about the various decisions that we made in order to run the experiments and the set of assumptions that made to run the experiments.

Hardware and Operating System

Due to the large training time involved we used a cluster of Linux nodes to distribute the task among multiple machines. The total number of machines being used at any given time changed according to availability, but it ranged between 5 to 26. Each of the machine was a Intel Xeon X5680 CPUs running at 2.33 GHz with 8GBs of RAM.

For Maxine VM

We conducted our experiments on a collection of SunFire 4150 machines. Each machine had two quad core Intel Xeon E5345 CPUs running at 2.33 GHz with 40GBs of RAM.

For Java HotSpot VM

We conducted our experiments on machines with Intel Xeon X5680 CPUs running at 2.33 GHz with 8GBs of RAM.

Compiler

We used compilers from two different VMs. The first compiler was the C1X compiler from the Maxine VM. The Maxine VM is an open source research VM developed by Oracle Inc. and is almost entirely written in Java. The second compiler was the server compiler from the Java HotSpot VM. The Java HotSpot VM is the most commonly used Java Virtual machine in the real world as is also the most commonly used in production environments. The VM internally comes with the client compiler

and the server compiler. The *server compiler*, is the version that is used when the same code is to be executed more number of times. In situations where the loading time of a code is less important than the steady state running time of a specific code.

4.9 Benchmarks

We used a total of five benchmarks suites for training and testing: the Java Grande benchmarks [57, 1], SPECjvm98 [50], SPECjvm2008 [51], SPECjbb2005 and DaCapo [58], and Scala Benchmarks [52, 59, 60]. The *Training Set*, is the set of benchmarks that are used during the training run of the machine learning algorithm. During this phase the neural network is being trained and the benchmarks would need to be executed many thousands of times. This means that the benchmarks would need to me small and quick in execution in order to create run through enough number of simulations. The *Test Set*, is the set of benchmarks that the machine learning algorithm has never seen before, and is used for actually evaluating the efficacy of the final solution proposed by the machine learning algorithm. This set is kept different from the *training set* in order to avoid over fitting. Another way of dividing the training set and the test set is called as the *n fold cross validation*, this is also discussed in greater detail in Chapter 2.

Java Grande

We use the *Java Grande* suite of benchmarks in the training set of the method inlining experiments on the Maxine VM. The Java Grande benchmarks that were used are listed in the Table 4.3 below.

The Java Grande benchmarks take in three input sizes, SizeA, SizeB, and SizeC which are in the increasing order of the amount of computation required. In our experiments we use SizeB, as it provided the right mix or quick execution and low noise.

Benchmark Name	Description
JGFCryptBench	Encryption and decryption library
JGFFFTBench	Performs Fast Fourier Transform
JGFHeapSortBench	Provides the implementation of Heap Sort for integers.
JGFLUFFactBench	Calculated Factorial
JGFSeriesBench	Fourier coefficient analysis
JGFSORBench	Successive over-relaxation
JGFSparseMatmulBench	Performs sparse matrix multiplication

Table 4.3: List of benchmarks in the *Java Grande* [1] benchmark suite.

SPECjvm98

We used the *SPECjvm98* suite of benchmarks as the training sets for both the Maxine VM as well as the HotSpot VM. The *SPECjvm98* suite is comprised of multiple set of individual benchmarks or applications presented in Table 4.4

Benchmark Name	Description
jess	an implementation of the Expert System Shell in Java
jack	A parser generator
raytrace	Provides an implementation of the raytrace algorithm.
db	A small implementation of a database with CRUD functionality
javac	Compiles a given piece of Java code into bytecode.
compress	performs compression
mpegaudio	Provides an implementation of the audio codec

Table 4.4: List of benchmarks in the *SPECjvm98* benchmark suite.

TODO add the rest of the benchmark suites.

For Maxine VM

We divided the benchmark suites into two sets, a training set and a test set. The *Java Grande* and the *SPECjvm98* benchmark suites have the shortest execution times, making them most suitable for use as the *training set*. The average training time for NEAT was around four days making other machine learning techniques like *leave one out cross-validation* (88 days) or *n-fold cross-validation* (40 days, assuming 10 folds) impractical. For our *Test Set* we used a all the benchmarks from the SPECjvm2008

and DaCapo benchmark suites that we could successfully compile and run with Maxine VM.

For Java HotSpot VM

We use the benchmarks *SPECjvm98* and *SPECjvm2008* as our training set and the benchmarks *DaCapo*, *SPECjbb2005*, and *Scala* for our test set. Scala benchmarks are primarily used in the Java HotSpot set of experiments as Scala can be compiled to bytecode that can be executed by the VM, however since the bytecode was initially generated by a non Java source code, the bytecode that is eventually generated might look very different than the standard Java bytecode.

4.10 Results

In this section, we discuss the results of our experiments using various machine learning approaches applied to the problem of constructing method inlining heuristics and discuss their relative advantages and disadvantages. The Figure 4.9 shows the effect of using the three heuristics Genetic Algorithm, Decision Trees and Neural Networks in Maxine Research VM. The heuristic tuned by the genetic algorithm is the present state of the art technique available as an option to a performance driven application developer. The results shown in the experiments present an convincing argument to use decision trees and neural networks instead at the factory by the compiler developer. Here the SPECjvm2008 and the DaCapo benchmarks were used as the training set. The SPECjvm98, Java Grande and Java Jolden benchmarks were used as the test set.

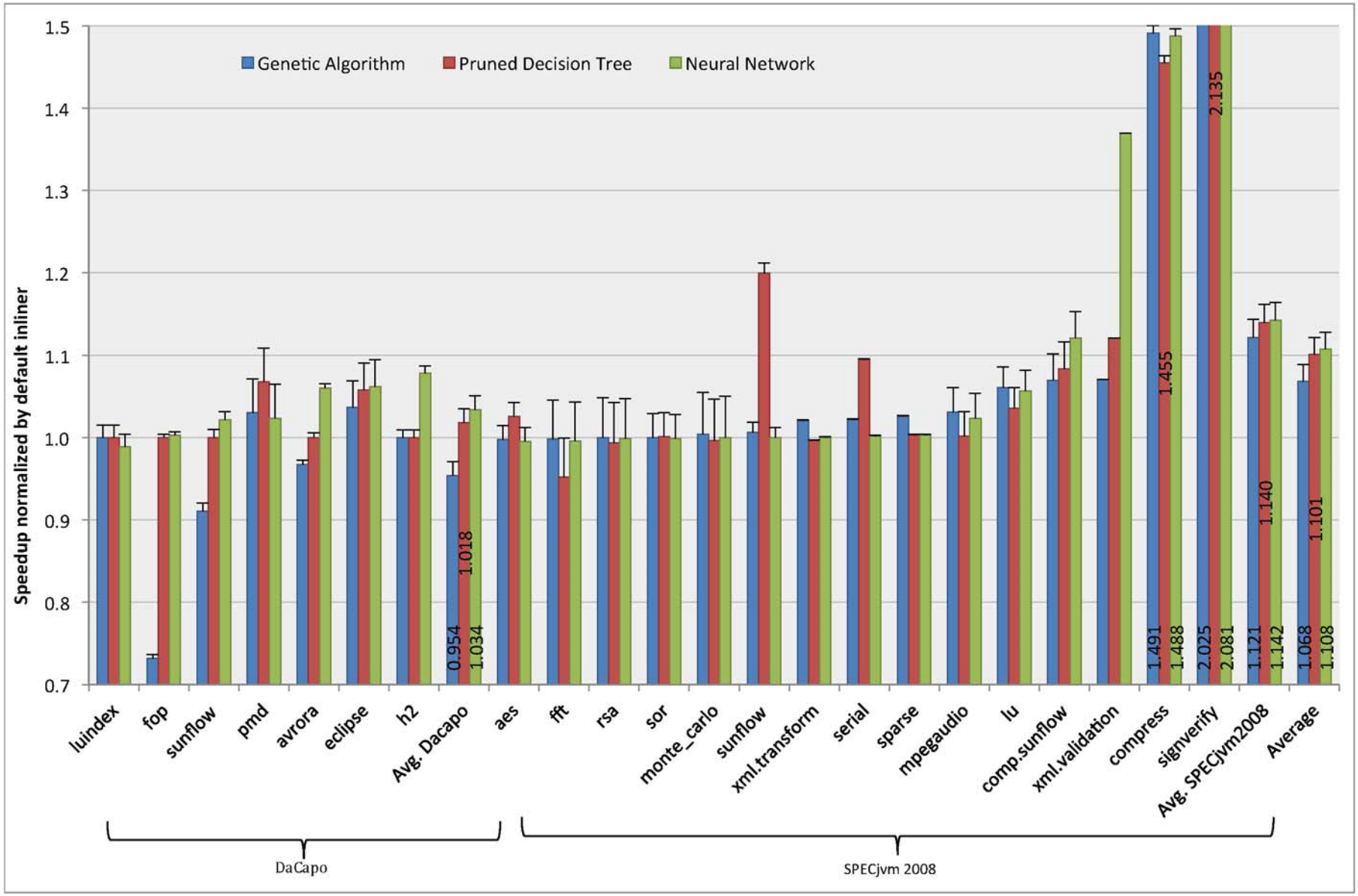


Figure 4.9: Comparative performance of method inlining using GA, Decision Tree and Neural Networks on MaxineVM

Results on Maxine VM

In our training experiments, the best network constructed in the first generation was 3% slower than the default inlining heuristic. However, NEAT quickly constructed effective neural networks to use as the inlining heuristic and in 50 generations the average speedup on our training benchmarks was around 25%. The performance of the networks constructed by NEAT stabilized after 250 generations and no significant performance gains were found after 250 generations.

The neural network that performed best on the training set was then used as the inlining heuristic in Maxine to compile the benchmarks in our test set, i.e., the subset of SPECjvm2008 and DaCapo benchmarks we could compile with Maxine VM.

Figure 4.9 shows that the best NEAT network achieved dramatic improvement for some benchmarks, up to 2.08 on signverify and 1.49 on compress. Note that none of the benchmarks tested had any significant slowdowns over the default heuristic. On average, the best NEAT neural network improved over the default inlining heuristic in Maxine by 11%.

Constructing Inlining Heuristics using Decision Trees

Though the neural networks constructed by NEAT perform well at inlining, it is difficult to obtain intuition from these networks. In order to construct more readable heuristics using machine learning, we decided to investigate the use of decision trees. We used the best NEAT network to construct the training data for the decision tree algorithm as described in Section 4.7. Once the training data was generated, we used the C4.5 decision tree algorithm in the Weka toolkit [56] to construct decision trees. We also automatically pruned the decision tree by limiting the height of the original decision tree. Note the heuristic constructed by the decision tree algorithm is significantly different from the default inlining heuristic shown in Figures 4.3 and 4.4. In particular, the decision tree algorithm found that it was important to use different

instruction types (e.g., conditional and unconditional branches) when deciding which methods to inline. The performance of the decision tree on the test set was 10% over the default inlining heuristic, which is comparable to the performance of the best NEAT neural network. This was an encouraging result, because our attempt at knowledge extraction from the neural network was successful. Figure 4.9 shows that we achieved significant improvement on several benchmarks in our test set using our pruned decision tree heuristic. In particular, we achieved improvements of 20% or more on `SPEC:sunflow`, `compress`, and `signverify`. There was no performance degradation for any of the benchmarks, except for fft which saw a degradation of 5%.

Unfortunately, the original decision tree generated by the Weka toolkit was very large and did not provide the concise and intuitive heuristic we were seeking. From information provided in the decision tree, we noticed that the original tree had over-fit our training set and a large number of branches were only being used to cover a small fraction of the training examples. We then pruned the decision tree to a fixed depth, and on evaluating over the training set we measured no major performance difference. The final decision tree is shown in Figure 4.10

The performance of the network when provided with three different types of inputs. The first bar labeled "Original Features" includes features used in the default inlining heuristic, the second bar shows the performance when the neural network is provided with all source code features mentioned in Table 4.2 except for Block Weight. The third bar is the performance when we used source code features as well as Block Weight. We used SpecJVM98 and SpecJVM2008 as our training set and DaCapo benchmarks and Scala Benchmarks as our test set.

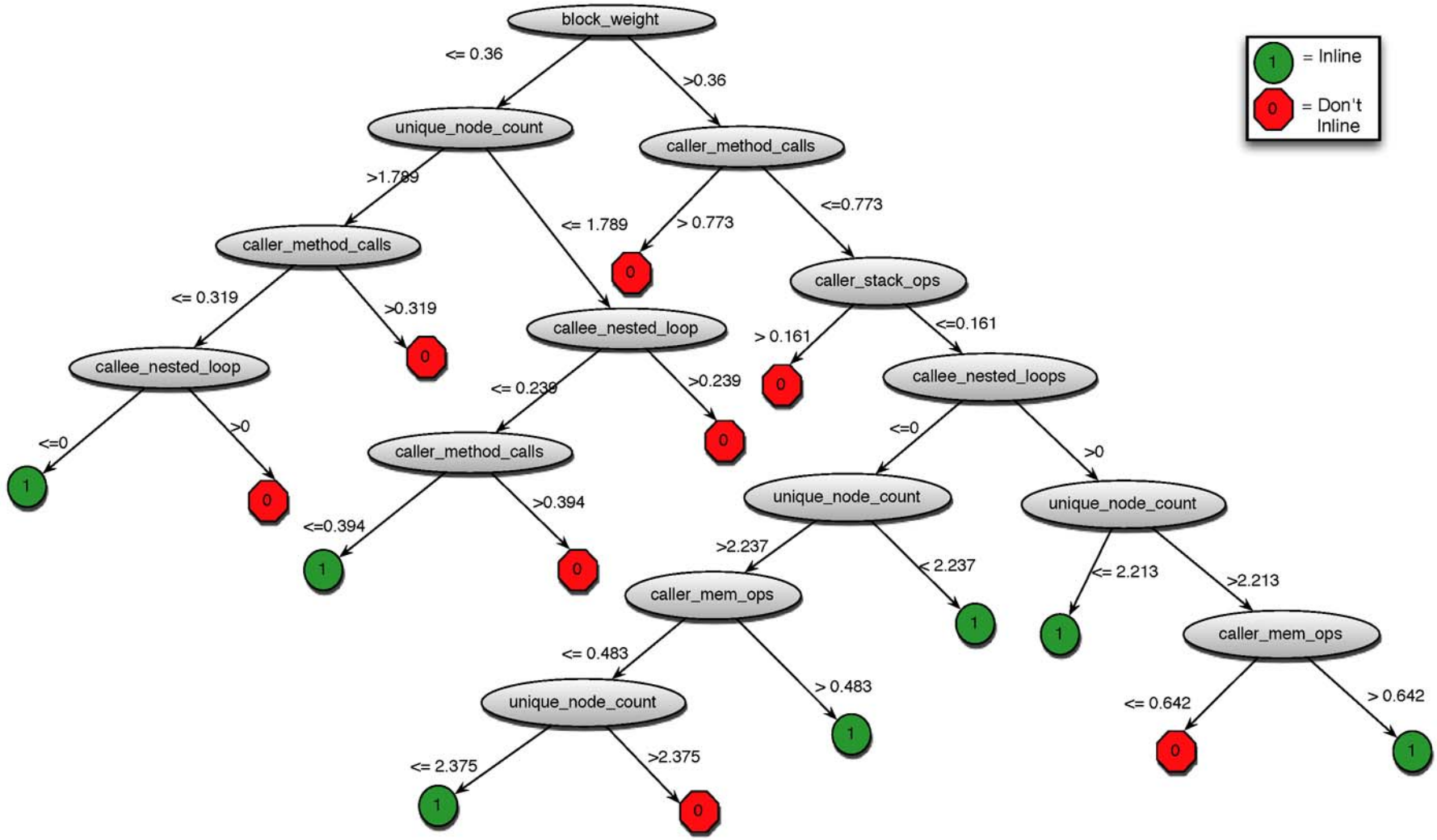


Figure 4.10: Decision tree generated from the trained ANN

Insights from Decision Tree

Similar to the insights that were found in the phase ordering experiments, using the neural network to generate a decision tree in the method inliner some useful insights. The decision tree generated for the Java HotSpot VM using static features and block weight is shown in Figure 4.10. From the decision tree that was generated, we could understand which features were most important for a very good inliner while making inlining decisions. The most important feature in our case was the block weight. This makes intuitive sense since the higher the probability of a method to be called the greater the advantage from inlining it. An interesting situation that we find is that if the method is nearing a threshold of not being inlined due to its size, the decision tree checks to see if there are a lot of memory operations. If there are a lot of memory operations, it still inlines the call perhaps assuming that inlining would help in improving instruction scheduling. Another source feature that seems to be fairly important is the number of other method calls in the method. This would predictably seems to adjust the importance of the present inlining decision. The importance of the present size of the method in making an inlining decision is fairly well established and can also be seen in the decision tree shown. The next most important factor in making and inlining decision was if the method call was present in a nested loop. Another parameter that seemed to be important is the concentration of memory operations. These particular parameters were never used to the best of our knowledge. Using the number of memory operations could suggest that it is more likely to see performance benefits if there are a lot of memory operations that might benefit from instruction scheduling at a later stage. Concentration of conditional statements could probably suggest that the machine learning algorithm was trying to put a weight on how probable a particular execution path would be in a method. More number of conditional branch statements in a caller would point to the fact that there is less probability of a given branch to be taken, thus making an inlining decision less effective. Given the height of the tree, we also see that the factors and the thresholds are dependent on other factors

Features	Feature Values	
	Default Values	GA-tuned Values
ALWAYS_INLINE_SIZE	6	6
CALLEE_MAX_SIZE	35	63
MAX_INLINE_DEPTH	9	11
CALLER_MAX_GRAPH_SIZE	8000	1888

Table 4.5: Default vs tuned heuristic for Maxine VM

and their thresholds. This is different from the present inlining heuristics that typically make decisions by considering one factor at a time. For example, it might make more sense for a medium-sized method to be inlined into a relatively large caller, if there are less number of conditional branches. On the other hand, the same might not be true if the caller has multiple *if-else* statements. Another situation could be that it might still be beneficial to inline if the callee has very simple code with few conditional branches.

Tuning Inlining Heuristics using Genetic Algorithms

Cavazos *et al.* [39] describe a method of tuning an existing inlining heuristic in a Java JIT compiler using genetic algorithms (GAs). We compared this technique with the approach introduced in this paper. Namely, we use GAs to tune the existing inlining heuristic in Maxine VM and compared this tuned heuristic to the new inlining heuristic constructed using decision trees. We used the ECJ toolkit [61] to tune the Maxine inlining heuristic using GAs. In order to fairly compare the results obtained with GAs with those obtained with NEAT, we used the same number of generations (250 generations) and generation size (60 individuals) for the GAs as was used with NEAT.

The best values found with the GAs provided an average speedup of 19.64% over the default inlining heuristic. These GA-tuned values are shown in Table 4.5. Figure 4.9 shows the performance of the GA-tuned heuristic on the test set. The average performance of the GA-tuned heuristic on the entire test set was 7% over

the default inlining heuristic. This is compared to the heuristics to the decision trees and NEAT, which achieved speedups over the default heuristics of 10% and 11%, respectively. Note that the GA-tuned heuristic performed poorly on two DaCapo benchmarks. It degraded the performance of `fop` by -27% and `DaCapo:sunflow` by -9% compared to the default inliner giving an average of -5% on DaCapo over the default.

However, a significant disadvantage of genetic algorithms is that it can only tune thresholds of features used in an existing heuristic and cannot construct a "new" inlining heuristic, or propose compound relations based two or more features. Comparing the results of NEAT and decision trees to GAs show that there is potential for improved performance when constructing an entirely new heuristic versus tuning an existing manually-constructed heuristic.

Results on the Java HotSpot VM

The Java HotSpot server compiler is one of the best tuned Java JIT compilers used in most production environments. We modified the HotSpot VM to perform similar machine learning experiments to show that our approach would be useful for even VMs that are highly tuned. We also experimented with different types of features. For the first set of experiments we used the same set of features that were being used by the default inliner. For the second set of experiments we added static source code features, and for the final set of experiments we added a profiling-based feature, which we call the "Block Weight". For the work on the Java HotSpot VM we used SPECjvm98 and SPECjvm2008 as the training set and then tested the final neural network on the DaCapo benchmark suite. We experienced a marked speedup on the `DaCapo:jython` of almost 90% when we used both profiling information as well as the static features. At present the Java HotSpot VM just like the Maxine VM uses a small subset static features to make inlining decisions as shown in Figure 4.4. In order to understand how good these features are we constructed three different NEAT heuristics using the

Benchmark	Neural Network	Pruned Decision Tree	Genetic Algorithm
DaCapo			
luindex	0.99	1	1
fop	1.00	1	0.73
sunflow	1.02	1	0.91
pmd	1.02	1.07	1.03
avrora	1.06	1	0.97
eclipse	1.06	1.06	1.04
h2	1.08	1	1
Avg. DaCapo	1.03	1.02	0.95
SPECjvm2008			
aes	0.99	1.026	1.007
fft	0.99	0.96	1.0
rsa	1.0	0.99	1.0
sor	1.0	1.0	1.0
monte_carlo	1.0	1,0	1.0
sunflow	1.0	1.20	1.03
xml.transform	1.000	1.0	1.07
serial	1.00	1.10	1.03
sparse	1.00	1.00	1.00
mpegaudio	1.02	1.00	1.02
lu	1.06	1.04	1.06
comp.sunflow	1.12	1.08	1.07
xml.validation	1.37	1.12	1.02
compress	1.49	1.46	1.49
signverify	2.08	2.14	2.03
Avg. SPECjvm2008	1.14	1.14	1.12
Cumulative Average	1.11	1.10	1.07

Table 4.6: Comparison of achieved speedup from ANN, decision tree, and the GA-tuned heuristic.

three different feature sets. The first bar labeled "Original Features" includes features used in the default inlining heuristic, the second bar shows the performance when the neural network is provided with all source code features mentioned in Table 4.2 except for Block Weight. The third bar is the performance when we used source code features as well as Block Weight. We used SPECjvm98 and SPECjvm2008 as our training set and DaCapo benchmarks and Scala Benchmarks as our test set.

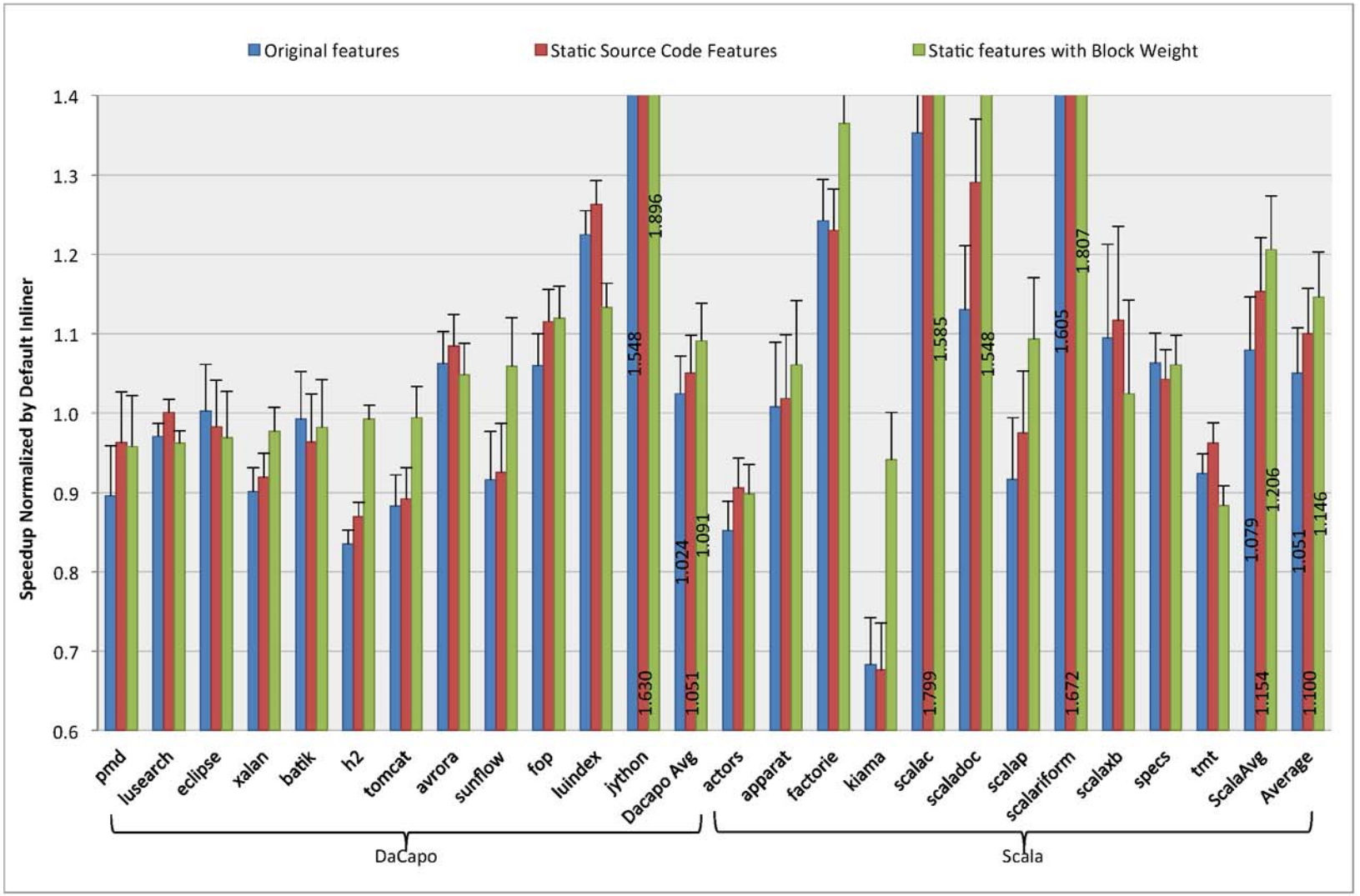


Figure 4.11: Speedup of DaCapo and Scala Benchmarks in the Java HotspotVM

Scala Benchmarks

Though originally the Java HotSpot VM was developed to run just Java programs, there are now several programming languages that compile the code to Java bytecode. Programming languages like Ada, Clojure, Groovy, JRuby and Scala target their code to run on the Java VM to take advantage of its portability and stability. Though the code being executed is still bytecode, the characteristics of these codes tend to be very different from bytecode generated for normal Java code. Scala Benchmarks is a collection of benchmarks based on the DaCapo benchmark suite. Using our method to compile these codes would test how our ANN reacts to code that might be completely different. The results are shown in Figure 4.11.

Original Features

The first bar in Figure 4.11 shows the final speedups found by the NEAT framework when using only the features that were used by the default Inliner. We were still able to get a small amount of speedup of 2.4% and 7.9% on the DaaCapo and Scala benchmarks, respectively. This suggests that the VM was better tuned for compiling normal Java code than code that was originally written in Scala. We see the same trend repeated for the other sets of features.

Static Source Code Features

This experiment utilized all the features listed in Table 4.2 except, for Block weight. When using these additional source features speedups were a little over 5% and 15% for the DaCapo and Scala benchmarks, respectively. This is shown by the second bar in Figure 4.11. Benchmarks `DaCapo:luindex` and `DaCapo:jython` improved by 26% and 63%, respectively. `Scala:kiama` had a slowdown of 67% but `Scala:scalac` and
`Scala:scalariform` also had very good speedups of 79.9% and 67.2%, respectively.

Static Source Code Features with Block Weight

This experiment included the block weight for the basic block of the call site. The Block weight present the Profiling based information that we talk in the paragraph 4.4 *Profiling Based Information*. This intuitively provides the best indication to the method inliner about the actual relative importance of each basic block of a method, which can me most valuable to the inliner as shown in the example shown in the Figure 4.6.

The best results amoung the different experiments were also were achieved when using block weight as one of the inputs to the method inliner. The average speedup over all the DaCapo benchmarks was just below 9%. Multiple Scala benchmarks have high speedups of more than 50%, like `Scala:scalac`, `Scala:scaladoc`, and `Scala:scalariform`, which improved by 58%, 54%, and 80%, respectively. The average improvement over all the Scala benchmarks were a little more than 20% of the baseline.

Performance on SPECjbb2005

The compiler compiles any source code provided to it to be compiled, and there is no possible superset of source code that can represent the set of possible codes that a compiler may be asked to compile. Benchmark suites are created with the goal of being able to emulate most common source code that the compiler might be asked to compile and generate binaries for. In the previous sections we divided a set of benchmarks into a training set and a test almost at random. The training set is used to train our machine learning algorithms, and the test set is never seen by our learning engine to tweak the final heuristic so that it may in any way customize the heuristic specifically towards the code seen in the test set. By creating this division we can, with reasonable confidence propose that the fitness achieved on the test set would correspond to performance improvements in the real world as well.

In this section we look at a special case of the SPECjbb2005 benchmark, and

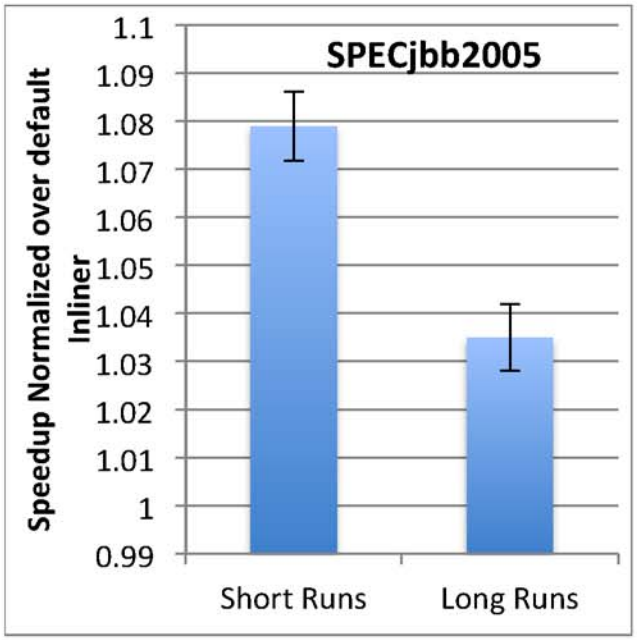


Figure 4.12: Speedup of SPECjbb2005 on Java HotSpot VM

we do not use a training set of a test set to present the results shown in 4.12. The HotSpot VM is the most commonly used production VM, and this also means a lot of effort is continuously spent on tuning it to to squeeze out all the performance from the VM. The developers of the Java HotSpot VM consider the SPECjbb2005 benchmark as the most important benchmark and present all the performance numbers of their VM with respect to the jbb benchmark. The VM developers also tune the VM using the SPECjbb2005 benchmark as well. Thus in the next experiment we propose to use our technique on the toughest VM using the toughest benchmark to see if our techniques can produce improvements in such demanding situations. The results published on the VM developer website tracking the best possible performance on the VM uses an optimized set of flags that tend to maximize the performance of the VM on this benchmark. We use the same flags for our baseline to show the effectiveness of our technique on even the benchmark that this VM is most optimized for.

For this experiment we used the short runs of SpecJVM2008 for the training set and show our results on the short runs and the long runs of SPECjbb2005 and

used all the source features shown in Table 4.2 including Block Weight. The short runs of the benchmark is run for 30 seconds for warm up and then run 16 times for 30 seconds each. The score provided by the benchmark is used as the actual result. For the longer runs the warm-up time is 240 seconds and then it is 16 time for 240 seconds each. The results for the experiment mentioned here is shown in Figure 4.12. The higher speedup of 8% above the baseline in the shorter runs could point to the fact that our method can method inlining decisions faster than the default inliner, and the default inliner would take longer to reach its optimum inlining settings. Even in the longer running tests we were able to achieve a small but very significant improvement of 3.5% which would still present huge saving over the extremely large number of the VM installations.

Chapter 5

OPTIMIZATION SELECTION

The final kind of compiler optimization discussed in this dissertation that could be used in proving optimization planning is optimization selection. In optimization selection, the compiler would select the right set of optimizations that could improve the performance of the code being compiled. During this research we use the GCC compiler and try an optimize a financial library being used by a commercial firm to model most of their quantitative needs.

We refer to the library in this chapter by the name *FLib* (Financial Library) and any regression tests or computational models that are used in the study are called $BM_1, BM_2 ... BM_n$. The computational needs of this library at present also constitute a sizable portion of their total computational needs. The goal of the dissertation was be to study the complete financial library and understand the computational needs. We looked at different machine learning techniques that could be used to optimize this library to provide a performance improvement. During this research we did not be use any source feature analysis as inputs to the machine learning algorithm, but instead used a more direct method of search space exploration using genetic algorithms.

During the course of this research we break the problem into four basic steps:

- optimization flag analysis
- search space exploration
- verifying the correctness of the best optimization sequences found
- creating a recommendation engine that can perform this search space exploration automatically

5.1 Introduction to Optimization Selection

Optimization selection is the process of selecting the most beneficial set of compiler optimizations that provide the most amount of benefit to the code being compiled. This selection of the optimizations that need to be applied to the compiler can be done at the time the compiler is designed (at the *factory*) or it could be done by the application developer at the point when the compiler is being used by the application developer to compile and create the application being developed.

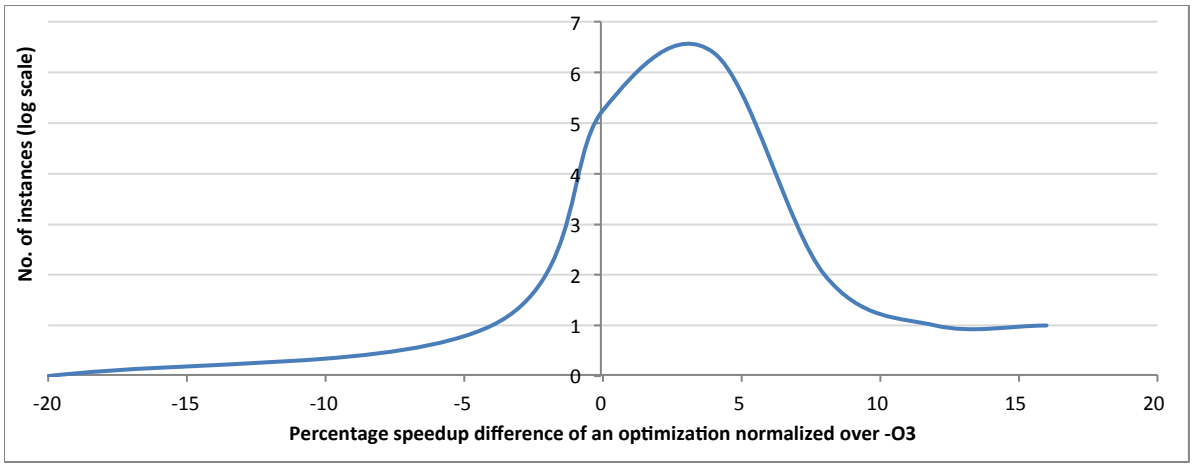


Figure 5.1: Frequency distribution of effect of a compiler optimizations on an application.

As discussed in the previous chapter modern compilers have a large selection of optimizations that are available to the compiler writer. There are three primary factors that effect the optimization selection process, and in the next paragraphs we talk briefly about all three of them.

Figure 5.1 shows the rough frequency distribution of difference in the performance of an applications as an effect of a specific optimization. On the y-axis is the log (base 2) of the number of optimizations that caused a specific shift in the performance of the code. The x-axis quantifies the amount of difference the optimization made in the performance of the code. For example if there are 32 optimizations that would make no difference in the performance characteristics of the code being compiled, corresponding

point is plotted at (0,5) on the Cartesian plane in the Figure 5.1. As we can see a large number of optimizations have no effect on the performance of the code. There is a very very small portion of optimization that has a large effect (either positive of negative) on the performance of the code being compiled. It is the job of the compiler writer and the application developer to select the right set of optimizations such that code that the compiler is most likely to see would be well compiled and performance would not be adversely effected.

Code being compiled

Optimizations are targeted at specific snippets of code. Only very rarely does one have an optimization that can produce performance improvement in all situations regardless of the code being presented. Newer optimizations are specific pinhole optimizations that take advantage of certain specific situations produced in the code. In most other cases the optimization would either have no visible effect on code being compiled or might even degrade performance for some codes. Effect of the optimizations present in the compiler vary with the code that is being compiled. This makes it important to change the optimization plan based on the code being compiled.

Optimization interactions

The compiler optimizations are applied on the code by the compiler one at a time, thus making it possible for indirect interactions between different compiler optimizations. Some optimizations are also dependent on each other, e.g. some are enabling optimizations that are designed to be applied after a specific set of other optimizations, or some could be clean up optimizations that need to be applied either periodically or based on past compiler optimizations applied. Due to such varied interactions that compiler optimizations have with each other, selection of optimizations are extremely interdependent. This makes the problem of optimization selection more difficult.

Architectural effects

Some optimizations take advantage of architectural shortcuts or architecture specific queues or caches to improve the performance of the code being compiled. In case the target architecture does not have these niche abilities, the tuned code might either have no effect on the code being produced or might produce code that might be worse than the original code. This dependence on the underlying architecture would need to be taken into account by the application developer when the software is being developed.

5.2 Optimization Levels

Optimization Levels in a compiler are the most basic form of optimization selection provided by the compiler. Setting different optimizations levels provide the application developer the ability to tune the code that is being developed to various degrees of aggressiveness. Optimization levels are flags that control the collective enabling or disabling of a group of optimizations. In GCC there are a few optimization levels O0, O1, O2, and O3 that are mentioned in more detail in Table 5.1. These optimization levels control more aggressive optimizations.

Looking at the optimization levels we see that the optimization levels are based on the probability of the optimization benefiting the code being compiled, this is the rudimentary way of performing a cost-benefit analysis of applying optimizations to a code.

One serious drawback of this approach is that the cost benefit analysis is done on the universal set of code that can be compiled by the compiler or the frequency with which the code might be seen by the compiler. Such assumptions would need to be made by the compiler writer at the time of compiler design or compiler development. This is a serious drawback, as the super set of all possible code is a lot more restrictive a constraint, and given a particular situation, or code or a library we can tailor the optimization selection to create a better performing code.

Optimization Level	Description
-O0	No code tuning optimizations are applied. Primarily used for initial development or debugging of code
-O1	Basic optimizations are performed
-O2	Most optimizations are enabled, optimizations that might not produce IEEE compliant approximations are not enabled.
-O3	Turns all all optimizations that can produce safe repeatable code, including inlining, vectorization etc.
-Os	Optimize for size, collection of all O2 optimizations that do not increase code size.
-Ofast	most aggressive optimization level that enables all optimizations, even the once that are not valid for standard compliant programs.

Table 5.1: List of Optimization levels in the GCC compiler.

5.3 Optimization Flag Filtering

The compiler that we used for the optimization selection experiments was the GCC compiler version 4.8. The GCC 4.8 compiler has large selection of optimization flags that could be used to optimize and tune the code being compiled.

In order to reduce or filter out the most effective optimizations we hypothesized that the effect of a single optimization could be a good approximation for the effectiveness of the same optimization in a group of optimization. This hypothesis does have flaws in ignoring the inter-optimization interactions, but we felt that it was a good way to reduce the total number of optimizations that we control. Reducing the number of optimizations under our control would reduce the search space of possible optimization configurations considerably.

The Figure 5.2 shows the result of just such an experiment. On the Y axis we plot the speedup normalized to O3, this is explained in the formula below.

$$S_n = \frac{t_{O3}}{t_{opt1}}$$

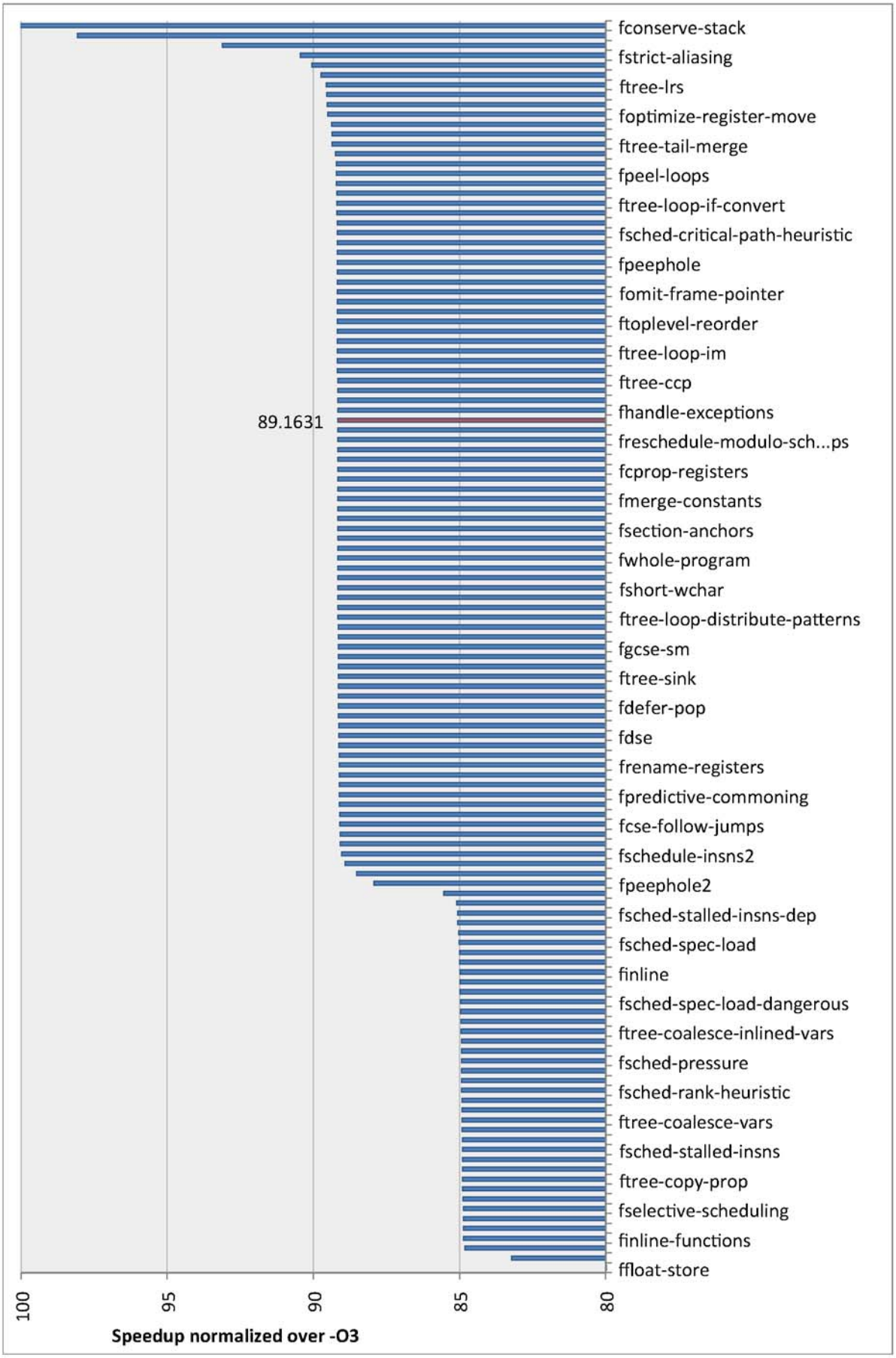


Figure 5.2: Performance of different optimizations on different benchmarks.

h

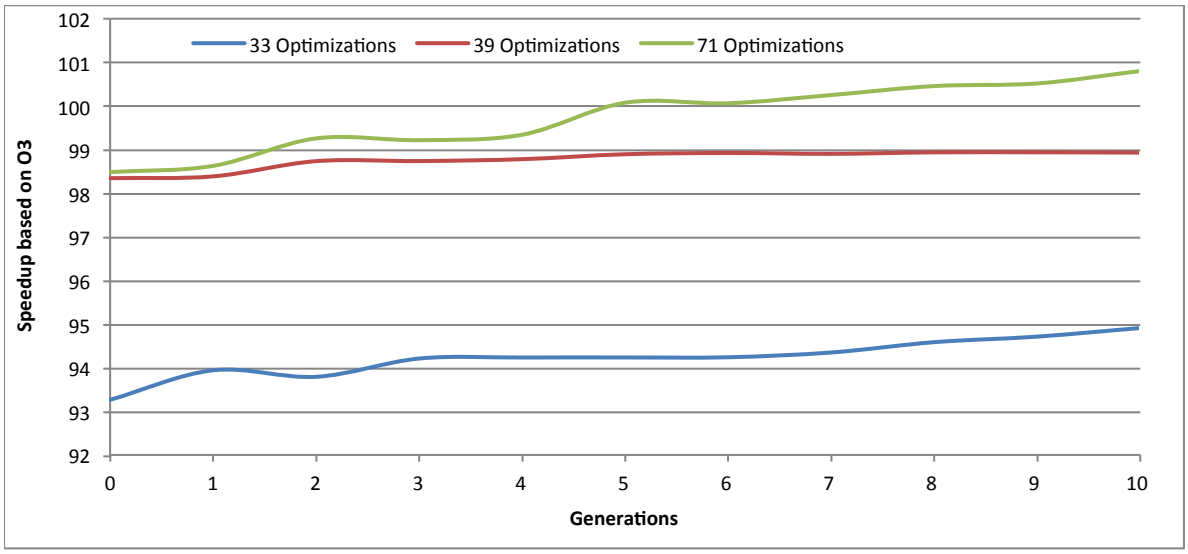


Figure 5.3: Performance of Genetic Algorithm when changing the number of optimizations in the search space.

Where S_n is the speedup normalized over O3, t_{opt1} is the execution time of the binary when compiled with the optimization $opt1$, and t_{O3} is the execution time when the benchmark is compiled with default $O3$ optimization level.

On the X axis we show the speedup that each version of the binary achieved when compiled by the optimization labeled on the X axis. The red bar is the performance when no optimization was applied to compile the binary, so each bar on the right of the red bar did better than having no optimization, and bars on the left actually degraded the performance of the application being compiled.

We used Genetic Algorithm to search for good optimization configurations. A more detailed explanation about Genetic Algorithm and the different parameters that we could use to tune it are explained in Section 2.11. Based on this experiment we were able to compile a list of 71 optimizations that were most effective.

The Figure 5.3 shows the different number of optimizations that we used in

the Genetic Algorithm. Initial experiment was done with a set of 33 optimizations and used Genetic Algorithm to find the optimization configurations that gave the best speedup or performance. The speedup achieved with 33 optimizations remained around the 95% of the baseline performance after 10 generations as shown in the blue line in the Figure 5.3. Increase in the number of optimizations increased the effectiveness of the GA by allowing it to control more optimizations, with 39 optimizations after ten generations the speedup was almost 99% of the baseline. We further increased the number of optimizations that were controlled by the GA to 71 different optimizations and the speedup achieved by the GA over ten generations increased to 101% of the baseline. This means that we had a 1% reduction in the average running time of all the benchmarks used in the GA. Based on this speedup we decide to use 71 optimizations for the GA experiments.

The final set of optimizations that were used for the GA experiments are presented in the Tables 5.2, 5.3, and 5.4. The first column of the table gives the name of the flag as used to activate it as a command line argument. The second column gives a short description of the optimization and what it aims to do. If the optimization almost always improves the performance of the code it is likely to be turned on in the lower optimization levels. However if the optimization has some side effects or may cause certain performance regressions in certain cases then it is not likely to be turned on by default.

Optimization	**Description**
-falign-functions	aligns function to optimal byte boundary
-falign-jumps	aligns jumps to optimal byte boundary
-falign-labels	aligns labels to optimal byte boundary
-falign-loops	aligns loops to optimal byte boundary
-fcaller-saves	saves register values to improve context switching
-fconserve-stack	minimize stack usage
-fcrossjumping	combines equivalent code to reduce code size
-fcse-follow-jumps	Continues CSE beyond *if-else* jumps
-fcse-skip-blocks	Continues CSE over basic blocks
-fdelete-null-pointer-checks	Removes null point checks improving code flow and branch prediction
-fdevirtualize	Convert calls to virtual function to direct calls when possible.
-fexpensive-optimizations	Perform minor optimizations them might be expensive in running time.
-fgcse	Global common subexpression elimination
-fgcse-after-reload	another load elimination pass to remove redundant spilling
-fgcse-lm	rearranges load-store combinations around loop boundaries to improve performance
-fhoist-adjacent-loads	Moves load operations to improve memory performance.
-findirect-inlining	perform inlining after tr
-finline-atomics	governs the inlining of *_atomics* routines.
-finline-functions	enables inlining
-finline-functions-called-once	modifies inlining aggressiveness for methods called only once.
-finline-small-functions	Enables inlining of small functions
-fipa-cp	perform intra-procedural(ipa) copy propagation
-fipa-cp-clone	Performs function cloning to create multiple images of functions with customized inputs
-fipa-sra	performs scalar replacement of aggregates
-fivopts	perform induction variable optimizations (e.g. strength reduction)

Table 5.2: List of GCC Optimizations used in optimization selection [2].

Optimization	Description
-fsched-critical-path-heuristic	Enable the critical-path heuristic in the scheduler.
-fsched-group-heuristic	Use the group heuristic for scheduling
-fsched-interblock	schedule instructions across blocks.
-fsched-spec	Move non-load instructions to improve code performance
-fsched-spec-load	Move load instructions to improve code performance
-fschedule-insns	Reorder instructions to eliminate execution stalls due to required data being unavailable
-fschedule-insns2	Similar to -fschedule-insns, but requests an additional pass of instruction scheduling after register allocation has been done.
-fshort-enums	Allocate just enough space to enums as required to store data regardless of binary compatibility of final code
-fshort-wchar	treat a *wchar* as a *short unsigned int*
-fstrict-aliasing	Assume strictest aliasing rules.
-fstrict-overflow	Assume strictest overflow rules.
-fthread-jumps	Checks if multiple consecutive if-else statements can be combined and or compounded.
-ftrapping-math	Compile code assuming that floating-point operations cannot generate user-visible traps.
-ftree-dce	Perform dead code elimination on trees
-ftree-lrs	Performs live range splitting in SSA trees
-ftree-partial-pre	Make partial redundancy elimination (PRE) more aggressive
-ftree-pre	Perform partial redundancy elimination (PRE)
-ftree-slp-vectorize	Perform basic block vectorization on trees
-ftree-sra	performs scalar replacement of aggregates in trees
-ftree-switch-conversion	Perform conversion of simple initializations in a switch to initializations from a scalar array.
-ftree-tail-merge	Search and replace identical code sequences
-ftree-ter	Perform temporary expression replacement
-ftree-vectorize	Perform loop vectorization on trees
-ftree-vrp	Perform Value Range Propagation on trees.
-funswitch-loops	Move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches
-fvar-tracking-assignments-toggle	Annotate assignments to improve debug information while optimizing.
-fvect-cost-model	Enable cost model for vectorization.
-fwrapv	integer signed overflow wraps

Table 5.3: List of GCC Optimizations used in optimization selection [2].

Optimization	Description
-fjump-tables	generate jump tables for switch statements
-fnothrow-opt	modifies throw statement handling to improve performance
-foptimize-register-move	Optimizes register motion
-foptimize-sibling-calls	Optimizes sibling and tail recursion
-fpartial-inlining	in-line small parts of a function, instead of the whole function
-fpeel-loops	Uses profiling information to flatten of peel loops when possible.
-fpeephole2	Control machine specific peephole optimizations.
-fpredictive-commoning	Tries to reuse memory loads/stores or previous computations.
-fprefetch-loop-arrays	Generate instructions to prefetch memory to improve performance.
-fregmove	move register numbers to improve register tying.
-frename-registers	Reduce false dependencies by improving register allocation
-freorder-blocks	Performs reordering of blocks to improve code performance
-freorder-blocks-and-partition	Performs reordering of blocks and partitions to improve code performance
-freorder-functions	Performs reordering of functions to improve code performance
-frerun-cse-after-loop	Performs another parse of CSE after a loop code to improve code performance
-freschedule-modulo-scheduled-loops	Use modulo scheduling for code in loops
-frounding-math	Performs number approximation to reduce running time.
-frtti	Enables support for run-time type information

Table 5.4: List of GCC Optimizations used in optimization selection [2].

5.4 Benchmark Selection

Compiler optimizations can just modify the way the final machine code is generated, however to quantify the effectiveness of the compilation it is important to have a set of codes that can be used to test and record the improvements. In the phase ordering and method inlining optimization experiments the assumption was such that the compiler would be modified at the *factory* and used in the real world. In this kind of environment the compiler could need to be tuned and fixed before the compiler gets a chance to see the real world code. In such situations the best way to optimizing the compiler and quantify the improvements would be to use a *training set* and a *test set*. In the present situation the compiler being tuned is actually being tuned to specifically compile one particular library. Quantifying the improvements on this library no longer need a *training set* and a *test set*, instead we just use the code from the library to create the baseline and compare it to the optimized code.

The library being studied has a very large number of test scenarios (more than 70,000) we could use a set of these to generate performance characteristics. We used three different criteria for selecting the benchmarks:

- Short running benchmarks
- Important benchmarks
- Low Noise

Short Running benchmarks

In order to quantify the performance of the optimization configuration, the genetic algorithm needs to compile and then run the compiled code. This process is repeated many thousands of times. If the benchmarks are long running then this approach of search space exploration would no longer be practical. In order to make the experiments short running and easy to run we limited the running time of the benchmarks to under 30 secs.

Important Benchmarks

Another important factor in selecting the benchmarks was the importance of the benchmarks. There is little to be gained form optimizing a piece of code that is no longer being used or is used infrequently, at the same time increasing the efficiency of a code that is frequently called would lead to bigger performance gains.

The library writers and the application developers that use the financial library were asked to point to the pieces of code that were important or slow running. We used this input to narrow our search to the more beneficial areas of the library.

Low Noise

Having low noise is critical in the accurate measurement of the performance gains achieved due to an optimization configuration. If the noise in the execution time is larger than the gain produced by the optimization configuration, it would be impossible to isolate the effect of the optimization configuration from the noise. Another way to look at this problem is if the noise of a particular benchmark is extremely low then it makes it easy for the Genetic Algorithm to attribute any performance gains or slowdowns to the compiler optimization configuration.

The final set of benchmarks that we selected were 11 in total from different parts of the library. The actual names of the benchmarks have been obfuscated to keep some of the information confidential.

5.5 Training

In this section we talk about the training of the chromosomes that encode the Genetic algorithm and the search space. The fitness of the GA is measured by measuring the performance of the benchmarks being compiled. The performance of the benchmarks are in turn measured using Dynamic Instruction Counts (DIC) instead of execution time, we talk about the advantages of using DIC over execution time in

Section 5.6.

Search Space

The search space that is presented by the application depends on the compiler as well as the code being compiled. Since we are trying to optimize a single library with static codes we do not have to worry about the search space being modified by the source code but just the optimizations available from the GCC compiler. In these set of experiments we use the optimizations provided by the GCC compiler. There are two primary types of optimizations:

1. *Binary Optimization* A binary optimization is an atomic optimization that can be turned on or turned off. There are no more fine grained used controllable parameters that can tune the behavior of the optimization e.g. Common Sub-expression Elimination.
2. *Tunable Optimization* Tunable optimizations are optimizations that can be tuned or modified by the user. This can be done to modify or tweak the aggressiveness of the optimization or it could be used to modify the behavior of the optimization in some other way, e.g. Method Inliing or Loop Unrolling. These optimizations are more complex and may contain small search spaces in themselves.

The magnitude of entire compiler optimization search space is extremely large, which is a common problem in compiler optimization planning. In order to understand the scale of the search space let us take the example of the GCC compiler. GCC has over 260 tunable compiler optimization flags. Tunable optimizations add extra complexity. So for the sake of this example we consider all optimizations to be just binary. If we assumed all the optimizations to be binary there would be a total 2^{260} possible optimization settings. With such a large set of possible optimization configurations, using brute force is not practical. Even sampling the search space does not provide us with a good coverage of the search space, and thus machine learning becomes the only option available to us. It is imperative that any search strategy used performs intelligent search space exploration. We perform this search space exploration using a

genetic algorithm as it has been shown to efficiently find good optimization sequences in large compiler optimization search spaces.

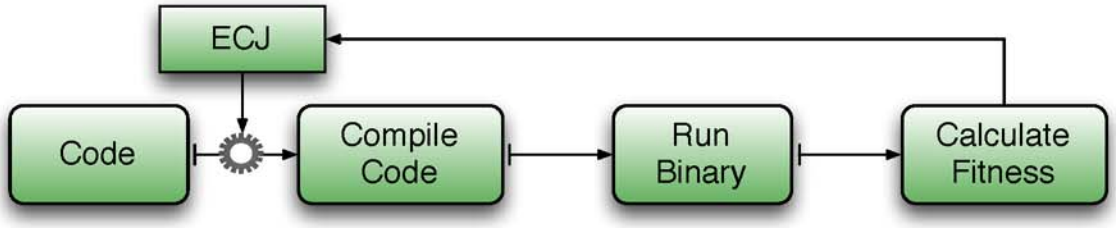


Figure 5.4: Block Diagram describing the use of Genetic Algorithm.

In our experiments we use the ECJ genetic engine to perform the search space exploration. The Figure 5.4 represents the steps involved in the searching process. The genetic algorithm used is an evolutionary approach to search space exploration. The algorithm starts with a random set of optimization configurations. These optimization configurations are then used to compile the set of benchmarks. The compiled benchmarks are then executed and the execution time is compared to the baseline compilation to calculate the fitness (or the relative performance speedup). This gives the algorithm a way to sort the optimization configurations in the order of their performance. The best performing optimization configurations are then used to generate new configurations for the next generation. This is repeated over multiple generations, and at the end the GA engine arrives on a really good optimization configuration. More information on the Genetic Algorithm, the theory and the different parameters used to govern the GA are explained in Chapter 2.

Numerical accuracy verification

The library being optimized is a financial library primarily responsible for mathematical calculations. Due to the nature of the calculations that are performed, the application is more sensitive to the accuracy of the calculations rather than the execution time. In order to maintain the numerical accuracy the library has a vast collection of regression tests that we could use. These regression tests are a collection of sample

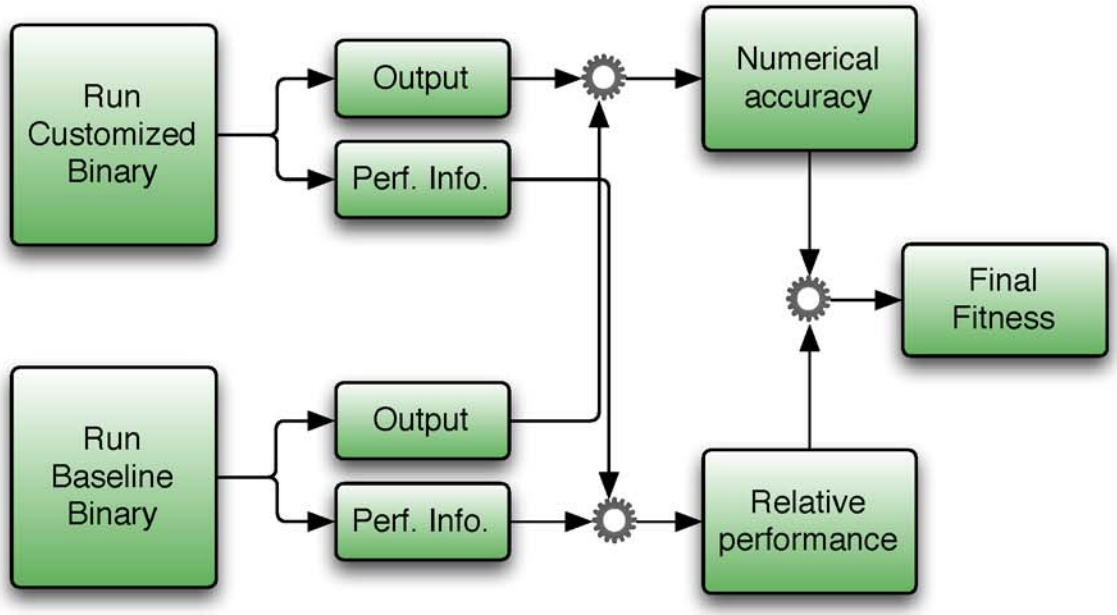


Figure 5.5: Composite fitness function combining execution time and numerical accuracy.

inputs and sample outputs. Running the library on these regression tests we collect the output generated by the library. The newly generated results are matched and compared to the sample outputs. We use these regression tests as a way of verifying the accuracy of the compiled binaries generated using our optimization configuration. Each numerical difference is penalized by reducing the performance by 10% relative to the baseline performance. This reduction in the performance helps the genetic algorithm to track and remember not to use optimization configurations that lead to machine code that result in numerical inaccuracies.

5.6 Dynamic Instruction Counts vs. Execution time

We used the valgrind toolkit to collect dynamic instruction counts to measure the performance. The actual parameter that we are trying to optimize is the execution time of the library, and in this section we present evidence that using DIC is a good analogy for the execution time.

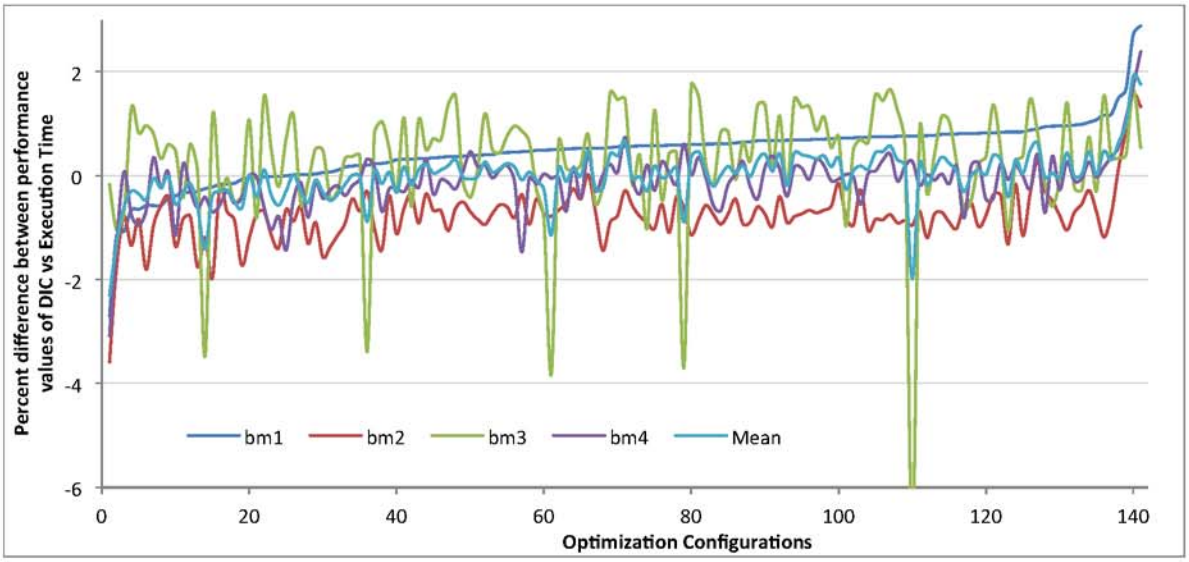


Figure 5.6: Difference in performance when measuring DIC and execution time.

In the Figure 5.6 we show the difference in the measure performance when using the two different strategies. We generated random optimization configurations and compiled the library using those configurations. Each image of the binary that was created using the optimization configurations were used to execute four different benchmarks. The performance characteristics for the execution were measured using the valgrind toolkit to measure the DIC, and simple execution time. We then calculated the difference in the two values obtained. The y-axis shows the actual difference in the measured performance. The x-axis represent the different configurations that we tested. The four lines bm1, bm2, bm3 and bm4 are for the four different benchmarks, and the blue line is the geometric mean of the four lines. The most noisy benchmark was the bm4 which had a maximum divergence from execution time of around 8%. For a large majority of the configurations (592 out of 600) the divergence was not more than 2%. The average divergence over all the 600 configurations that were tested was just 0.32%.

From this set of experiments we see that there is not a lot of difference in the difference in the two performance matrices. Thus any performance gained when using

the DIC as the training parameter is likely to also result in performance gaines when measured using execution time.

Random vs. Intelligent optimizations

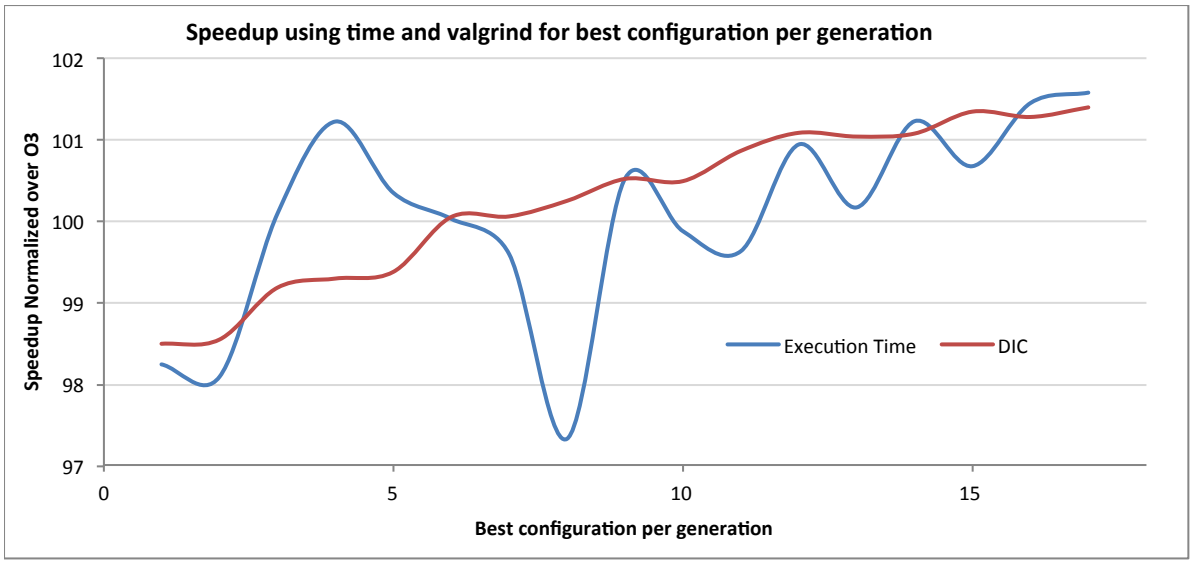


Figure 5.7: Difference in performance when measuring DIC and execution time.

Another concern that we had when presenting the results in Figure 5.6 was the fact that we were using random optimization configurations. There is a possibility that good optimizations might change the library in such a way that the machine code takes advantage of the underlying architecture of the machine in the form of vectorization or better use of the cache that might not be emulated accurately when using the valgrind toolkit.

In order to confirm our assumptions still remained accurate we used the best chromosomes found in each generation during the GA training to compile the library and run it in production environment just measuring the execution time. This would be an extremely accurate representation of the real world problem that we are trying to solve.

The Figure 5.7 represents this particular data. The x-axis represents the best performing chromosomes of each generation, and the y-axis shows the percentage relative speedup normalized over the baseline. Looking at the graph we can see that the performance gains measured using DIC (the red line in Figure 5.7) are in the increasing order, which points to the fact that the machine learning algorithm is in fact able to search for good optimization configurations, and also that over generations the configurations that it does find are progressively better than the previous optimization configurations when using DIC or execution time as the standard of measuring performance. This could be treated as conclusive proof for the environment of the present experiments, using DIC instead of execution time still provides us with good optimization configurations.

Reasons for choosing DIC

There are certain advantages of using DIC over execution time, that are explained in the paragraphs below.

Low noise

The machine learning algorithm is very sensitive to noise during training. The sole criteria to choose one optimization configuration over another is based on the measurement of performance characteristics. This is done by making very small changes and adjustments to the already existing configuration and measuring the difference in the performance. If the noise in the system is higher than the difference in the performance due to the change in the optimization selection, the machine learning algorithm would not be able to attribute the change correctly to the right factor. It is the ability to track and remember small performance changes and combine multiple such changes in the optimization plan that make it possible to use machine learning. It is absolutely critical that we use the method with least noise profile for this reason.

Multiple instances on one machine

The machines that we used as slave nodes in the cluster were dual processor machines with 16 cores in each processor. The benchmarks that we used to test the library were essentially single threaded applications that do not take advantage of multi cores. In order to take advantage of all the cores available on the system one would need to run multiple tests at the same time. Doing this had no effect on the performance numbers collected using DIC however the noise profile using execution time increased in direct relation to the number of tests running in parallel.

5.7 Experimental Setup and Terminology

The *optimization selection* experiments on the GCC compiler and the financial library were performed on a cluster of machines, governed by the master node that performed the genetic exploration. The master node would generate the chromosomes that needed to be evaluated, and these chromosomes would be shipped out to the slave nodes for further evaluation. The terms used here are explained below.

Chromosome

The term chromosome is used to refer to a vector that essentially encodes the optimization configuration. The chromosome is generated and evolved by the Genetic algorithm engine which uses the optimization configuration as genetic material that is evolved over multiple generations to improve the fitness of the configuration. This similarity to genetic material has lead to it being referred to as the chromosome. The chromosome as shown in the Figure 5.8 is an array of flags that are encoded using numbers. All the optimizations can be divided into three types.

1. *Optimizations encoded in Binary* These optimizations are just *on/off* flags that represent the presence of the absence of a specific optimization.
2. *Optimizations encoded in Integer* Optimizations that can be encoded into a specific tunable discretized value.

Figure 5.8: Chromosome used to control the optimization sequence of the GCC Compiler.

3. *Optimizations encoded in Float* Optimizations that can be encoded into a specific tunable continuous value. For ease of use we discretize the continuous values into appropriate steps.

4. *Optimizations encoding Specific Options* Optimizations that represent a selection of a single option from multiple options available. For example choosing one algorithm out of multiple choice of algorithms that can perform register allocation.

If the non binary optimizations have more than one tunable parameter then one can extend this work to encode information governing the optimization into more then one slot in the chromosome. However such complex optimizations were not a part of this set of experiments. This encoded array of flags as shown in Figure 5.8 are used to control the compilation and optimization process of the library when using the GCC compiler.

Master Node

The role of the master node as the central node in the cluster is to generate the chromosomes. This task is not computationally intensive, and thus the slowest machine was selected as the master node, represented by the center node in Figure 5.9. Once the chromosomes are generated the master node would then ship them for evaluation to the slave nodes. In our experiment we used an Intel Quad core processor with 4GB RAM running on Red Hat Operating system.

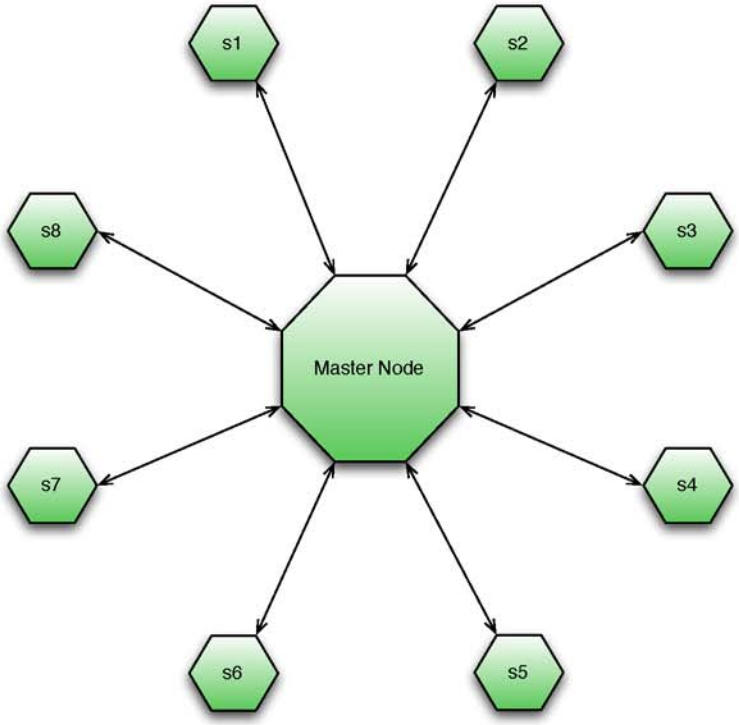


Figure 5.9: Representation of the GA cluster architecture.

Slave Node

The task of the slave node (represented by the outer nodes in Figure 5.9) can be primarily subdivided into three sub tasks described below. These tasks are also shown in the Figure 5.4.

1. *Compilation* The slave node uses the chromosome received from the master node to compile the financial library.

2. *Execution* Valgrind toolkit was used to measure the dynamic instruction counts (DIC) during the execution of the benchmarks to measure the performance. During the execution phase, the DIC is measured for each of the benchmark.

3. *Evaluation* The measured DIC is then compared to the baseline DIC's of each of the benchmarks and a relative speedup is calculated which is used to measure the effectiveness of the optimizations used. This effectiveness of the optimizations is used as a direct value to represent the fitness of the chromosome being evaluated.

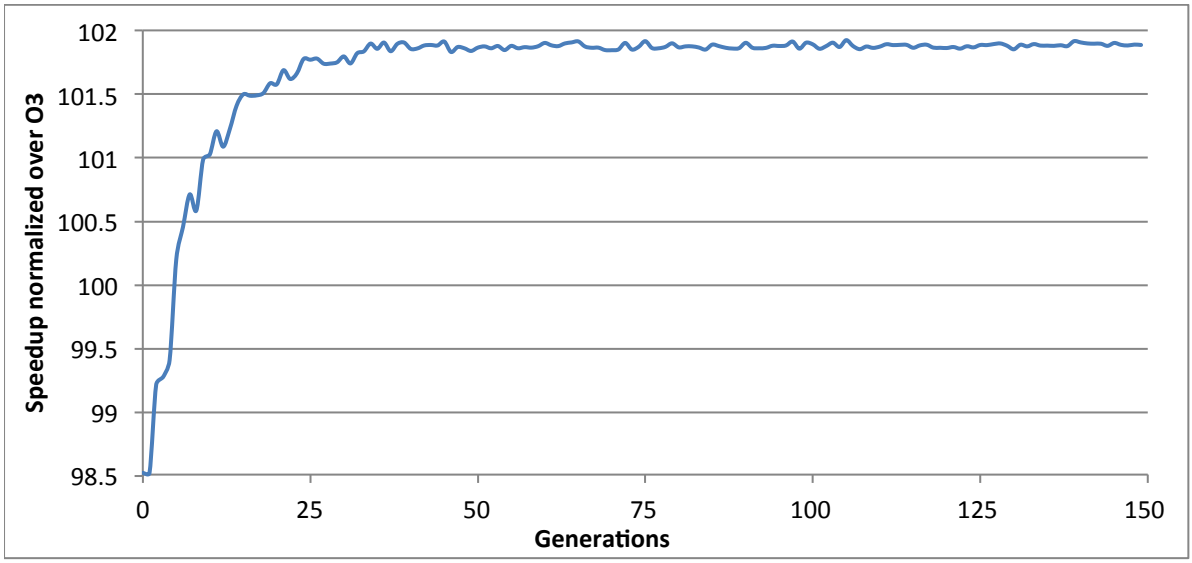


Figure 5.10: Evolutionary improvement of optimization configuration over generations.

5.8 Results

In this section we describe the effect of performing optimization selection on the financial library *FLib*.

Effectiveness of Machine Learning

The GA used for the search space exploration was very effective in searching for good optimization configurations. Over a period of 40 generations the ECJ engine was able to achieve a 2% speedup over baseline. We continued the search to give the ECJ engine a chance to find a better optimization configuration.

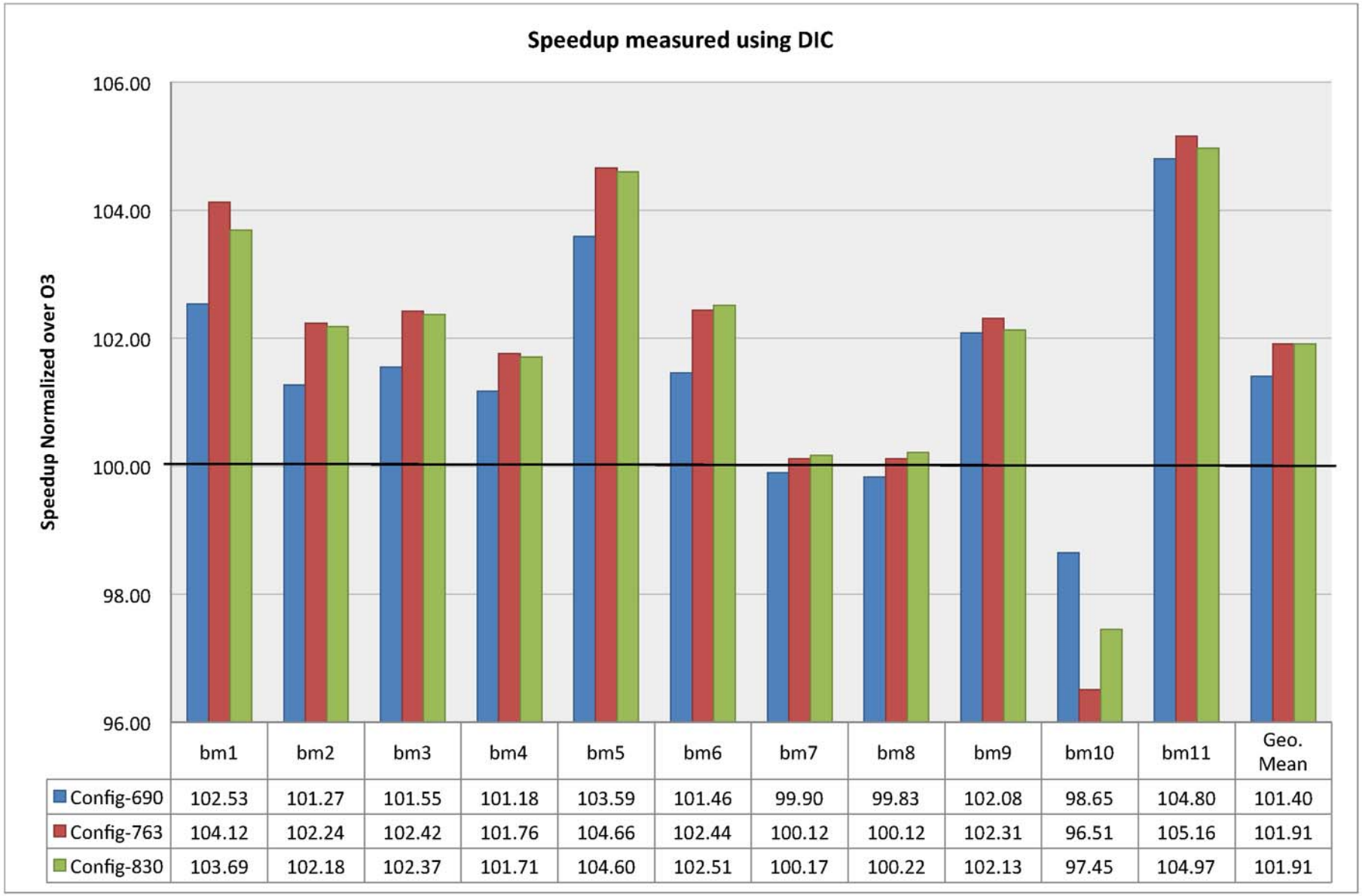
Graph shown in Figure 5.10 represents the evolution of the optimization configurations as searched by the ECJ engine. The x-axis represents each generation during the evolutionary cycle. The y-axis shows the speedup compared to the baseline. The baseline in this situation is the performance of the application when compiled using the O3 optimization setting.

Performance Gains

The performance measurement criteria provided during the learning phase using the ECJ engine was the Dynamic Instruction Counts (DIC). This gave the machine learning algorithm an accurate low noise parameter to perform the search space exploration. However the library developers and the library users would actually prefer to understand the performance difference in terms of execution time. Keeping this in mind the performance numbers are presented in both, execution time as well as DIC. Section 5.6 talks about the reasons for choosing DIC over execution count in detail.

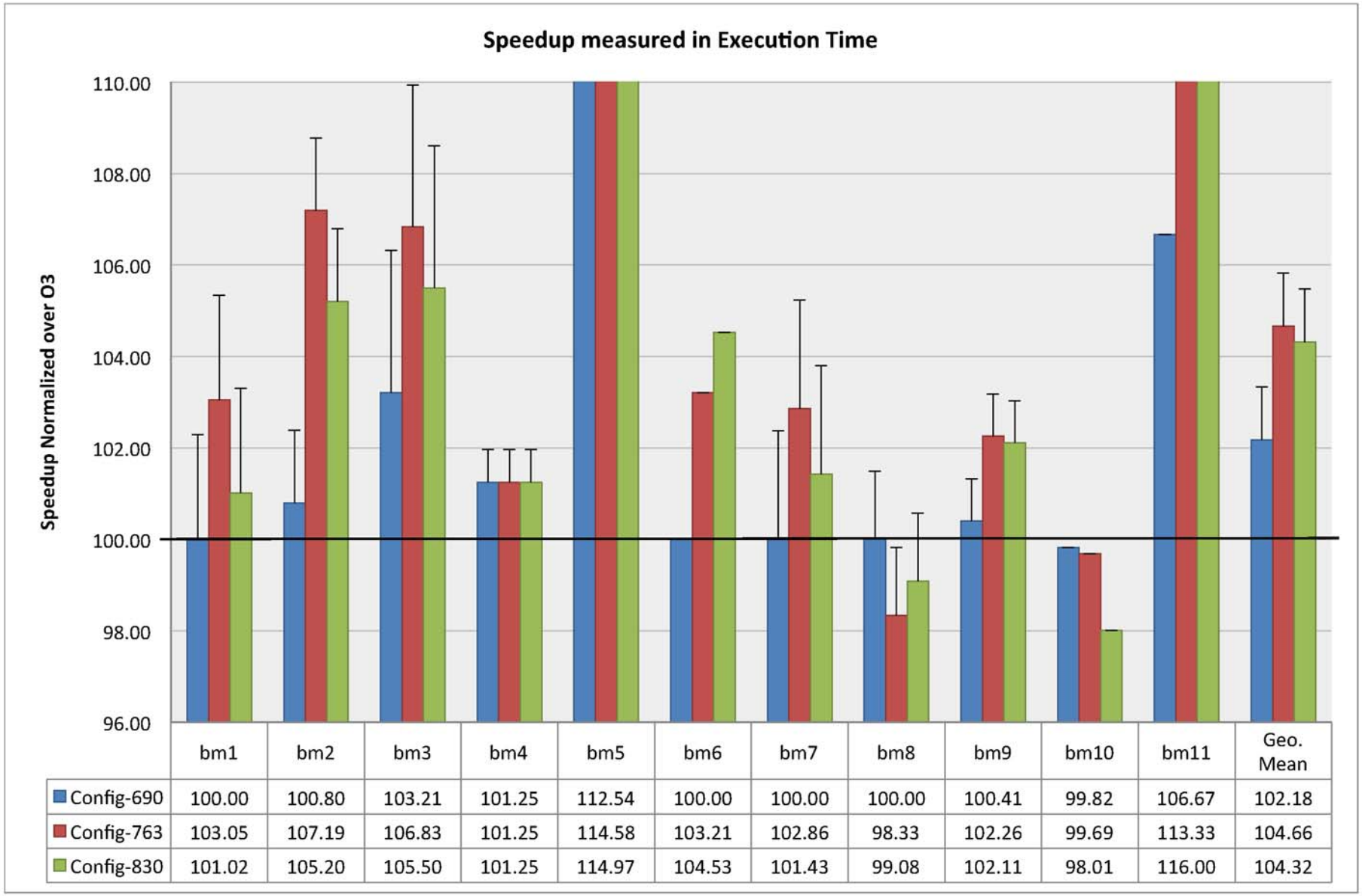
The Figure 5.11 shows the performance improvements of the three best performing chromosomes during the evolutionary search during the last few generations. The chromosomes achieve an average of 2% improvements, and in few cases as much as a 5% improvement. On the y-axis we have the performance improvement of the benchmarks shown using the speedup normalized over baseline. The baseline used in this case is the binaries obtained by compiling the source code using *-O3* as the optimization level. On the x-axis we have the obfuscated names of the benchmarks that were used to measure the performance. The name of the benchmarks have been obfuscated in accordance with the wishes of the financial library developers at J. P. Morgan.

The Figure 5.12 similarly show the performance improvements of the three best performing chromosomes when measured using execution time. The x-axis and the y-axis are similar to the graph in Figure 5.11. We use execution time as the matrix to confirm that the improvements that we gained from using DIC during training did infact carry forward to the real world situation of measuring execution time. We see an average improvement of a little more than 4%, and in the case of benchmarks bm5 and bm11 we get 15% and 16% improvement respectively. We show the error bars in black.



	bm1	bm2	bm3	bm4	bm5	bm6	bm7	bm8	bm9	bm10	bm11	Geo. Mean
Config-690	102.53	101.27	101.55	101.18	103.59	101.46	99.90	99.83	102.08	98.65	104.80	101.40
Config-763	104.12	102.24	102.42	101.76	104.66	102.44	100.12	100.12	102.31	96.51	105.16	101.91
Config-830	103.69	102.18	102.37	101.71	104.60	102.51	100.17	100.22	102.13	97.45	104.97	101.91

Figure 5.11: Speedup measured in Dynamic Instruction Counts.



	bm1	bm2	bm3	bm4	bm5	bm6	bm7	bm8	bm9	bm10	bm11	Geo. Mean
Config-690	100.00	100.80	103.21	101.25	112.54	100.00	100.00	100.00	100.41	99.82	106.67	102.18
Config-763	103.05	107.19	106.83	101.25	114.58	103.21	102.86	98.33	102.26	99.69	113.33	104.66
Config-830	101.02	105.20	105.50	101.25	114.97	104.53	101.43	99.08	102.11	98.01	116.00	104.32

Figure 5.12: Speedup measured in execution time.

Noise

The noise profile for the benchmarks when using execution time are represented by the noise bars in the Figures 5.11. The average noise when measuring execution time was 1.15%. The benchmark bm3 was the noisiest with the noise of 3.11%. Figure 5.11 does not have any error bars as the measured noise levels were 4 to 6 orders of magnitude smaller than the noise when the performance is measured using execution time.

$$
\begin{aligned}
noise &= \frac{stdev}{t_{avg}} * 100 \\
stdev &= \sqrt{\frac{\sum_{i=0}^{n} (t_{avg} - t_i)^2}{n}} \\
t_i &= time\ of\ i_{th}\ run \\
t_{avg} &= average\ time\ of\ i\ runs
\end{aligned}
\tag{5.1}
$$

The Figure 5.13 represents the noise in the system when we measured the performance of the system using execution time. The noise is measured by running the benchmarks 20 times and calculating the average running time of the benchmark. Once the average running time is calculated the standard deviation is calculated using the formula presented in Equation 5.1. The ratio of the standard deviation to the average running time gives use the percentage noise in the benchmark being used.This noise is also dependent on the load on the system as well, and thus when using execution time for performance measurement we only ran one timed experiment at a time to keep the noise to a minimum. In case of the GCC experiments when referring to average speedup we used the geometric mean of the values being collected.

Relative importance of optimizations

Not all optimizations are created equal. Some optimizations could be more effective in providing performance improvements than others. Also the effect of the

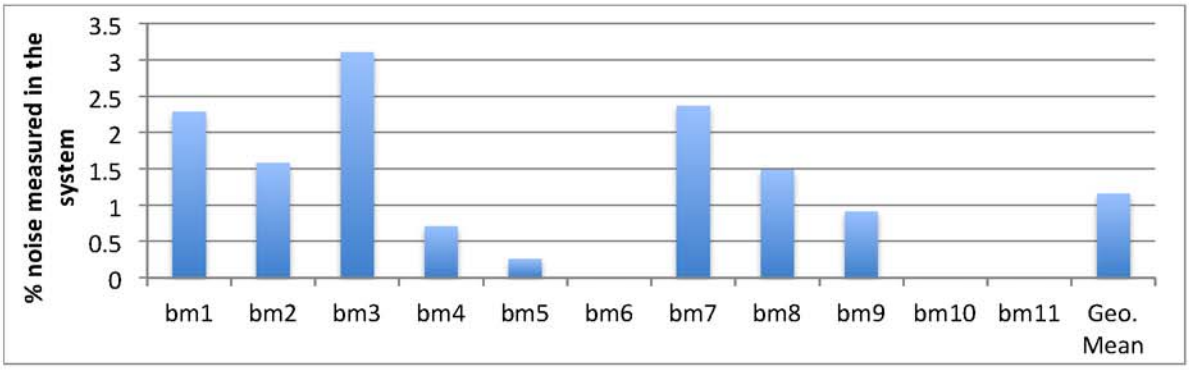


Figure 5.13: Average noise in system when measuring execution time.

optimizations might be change based on the codes that are being compiled. It is always difficult to understand intuitively which optimizations would be used to be applied and which might cause more harm than good. This makes the job of the application developer even more difficult. In order to solve this problem the application developer could use genetic algorithm to provide a good set of compiler optimization selections that would provide the most performance speedups when compared to the baseline. It would be useful to know the most effective set of optimizations filtered out from the universal set of optimizations.

The list of such compiler optimizations would be very useful to an application developer, and also a very interesting exercise for the compiler developer. The compiler writer would be able to understand first hand the way the optimizations are used and the effect they have on the codes being compiled.

The Figures 5.11 and 5.12 represent the speedups that were achieved using the best possible optimization selection. These speedups were achieved by the using a set of optimizations as discussed in this chapter. The Figure 5.14 presents the relative importance of the optimizations used in the optimization selection process.

We used machine learning to find the best possible set of optimizations from a set of 71 total optimizations. The Machine Learning algorithm went through those optimizations and selected the set of optimizations that were most useful and used

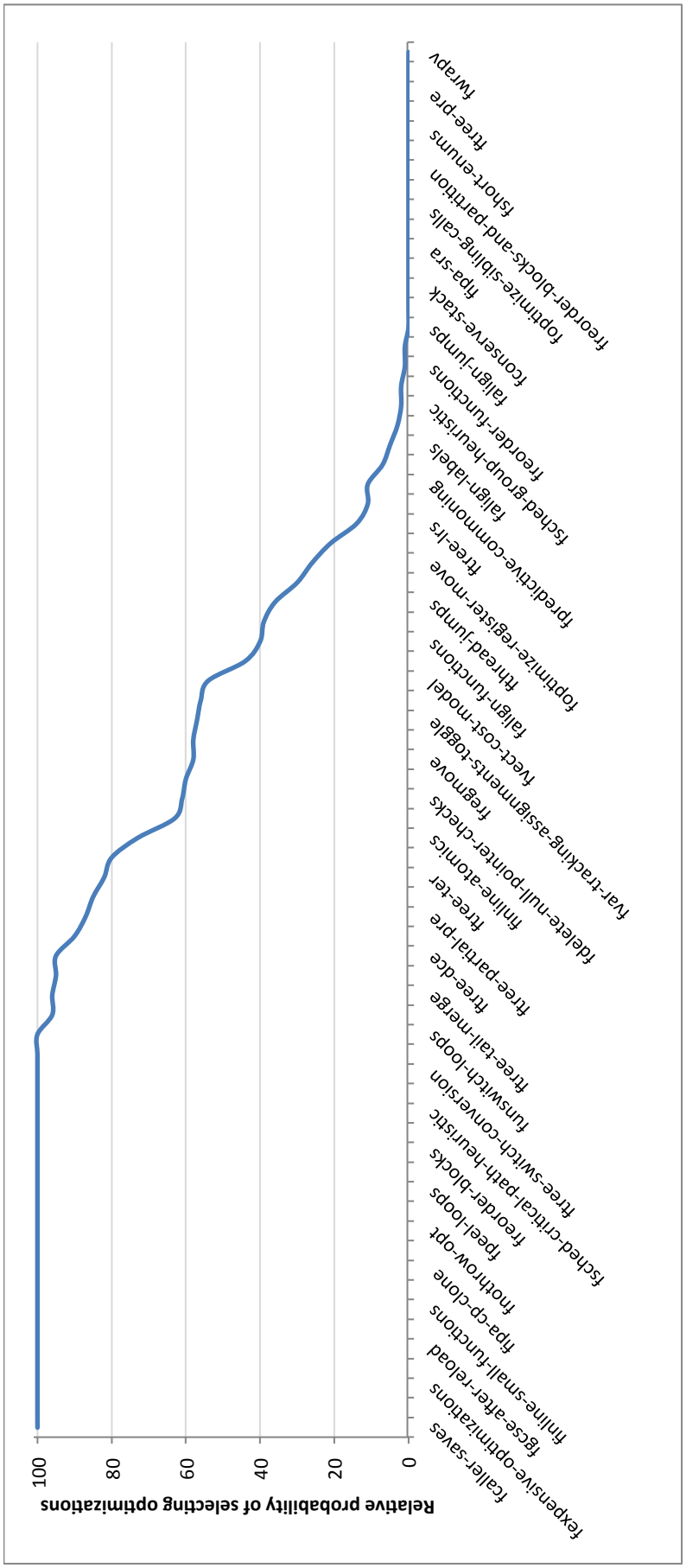


Figure 5.14: Relative importance of optimizations in the best performing optimization sequences.

them in the final set of optimizations. The optimizations that were most useful were reused and kept in the final set of optimizations. The optimizations that reduced the performance of the code would be actively removed by the genetic algorithm. This ability of the genetic algorithm to increase the probability of a useful optimization being present in the final set of optimizations and actively trying to reduce the probability of a harmful optimization can help us gain an insight on which optimizations are useful and which optimizations are not useful.

The Figure 5.14 shows the relative probabilities of different optimizations when selected by the genetic algorithm. The optimizations that were very useful are on the extreme left, and have a near 100% chance of being picked up by the ML algorithm. The optimizations that were not beneficial to the benchmarks but at the same did not cause any harm in reduced performance and in general did not statistically affect the performance of the application were neither removed nor actively added to the optimization selection, these optimization comprise the middle section of the figure 5.14. On the right side is the set of optimizations that statistically reduced the performance of the benchmarks and the application in general. These optimization were actively avoided and thus were not present in the final optimization sequences found by ECJ.

Chapter 6

CONCLUSION

During the course of this research we were able to demonstrate the use of machine learning on a broad range of compiler optimization problems. We looked at phase ordering, optimization tuning as well as optimization selection. These three forms of optimization customizations represent the complete range of modifications that one can perform using a compiler short of introducing a new compiler optimization.

In this thesis we have shown that using source code features in combination with different machine learning techniques provide a significant performance improvement for code being compiled. We present our results on improving method inlining and phase ordering in three different compilers, Maxine Research VM, Java Hotspot, and JikesRVM.

The primary focus of this research has been to use propose an method to provide customized recommendation of optimizations individually tailored to the code being compiled at each instance. We create customized optimization plan recommendations for source code being compiled by using the source code as a decision making factor in the machine learning algorithm. The source code is distilled into source features that try and represent the source code in a manner that would be useful as inputs to the machine learning heuristic. To the best of our knowledge, this is the first research to demonstrate that source features as inputs to machine-learning models can be successfully used to tune optimization parameters and dynamically choose a good ordering of optimizations for previously unseen methods. The present study is promising as it provides a fresh prospective to the problems of method inlining and phase ordering which have been studied for decades.

The research done in customizing phase ordering in Chapter 3 provided customized compiler optimization plans for a method using source features as inputs to an Artificial Neural Network. Using ANNs we were able to show an average of 8% improvement in the adaptive compilation scenario and 8.2% improvement in the optimizing compilation scenario. We achieved a maximum speedup of around 24% in mpegaudio. We were also able to reduce the average optimization sequence length by 25% presented in Table 3.5 which is a great result in a dynamic compilation environment where the compiler and the application being executed share the same resource pool. In Listings 3.1 and 3.2 we also take an example of a piece of code that was compiled using the traditional optimization compilation plan and compared it with the code generated using the plan generated using the ANN.

When using machine learning to tune Method Inlining on the Java HotSpot VM and the Maxine VM we achieved a 14% and 10% speedup respectively (in Section 4.10). We also explain in Section 4.7 our method of converting an unreadable ANNs into a much more understandable format of a decision tree. There are two more interesting results that we should highlight from this research. First we were able to achieve a small but a very significant speedup of 3% on the SPECjbb2005 benchmark (in Figure 4.12)using the Java HotSpot VM. The HotSpot VM is the most tuned and the most popular Java VM and the benchmark that is used primarily used to tune the VM is the SPECjbb2005 benchmark. Being able to achieve any speedup on this VM(Compiler) and benchmark combination we feel was a very promising result for our methodology. Secondly we were able to achieve significantly better results on the Scala benchmarks (in Figure 4.11)than the default compiler, which points to the robustness of the heuristic proposed by the machine learning algorithm. Our use of source features and profiling information in combination with different machine learning techniques to customize method inlining decisions is novel. The use of decision trees in choosing the right compiler optimization parameters is also new and holds a lot of potential for practical use in the real world.

Work in Chapter 5 provides a great example of how a machine learning and compiler optimizations can be combined to improve performance of a real world large scale library. We were also be able to study and document the different steps involved from selecting a good set of benchmarks to finding good optimization sequences. We were able to achieve an improvement of 2% to 4% speedup using optimization selection techniques presented in this thesis. Since we were optimizing a financial library, numerical accuracy was extremely important to the final user of the application, we implemented an automatic feedback mechanism that would evaluate and adapt the fitness of the final heuristic based on the numerical accuracy as well as the speedup of the final compiled code.

Based on the research done here we feel that the use of source features in suggesting customized optimization compilation plan is very promising and could be used more often in most real world compilation scenarios.

BIBLIOGRAPHY

[1] J. A. Mathew, P. D. Coddington, and K. A. Hawick, "Analysis and Development of Java Grande Benchmarks," in *In Proc. of the ACM 1999 Java Grande Conference*, 1999, pp. 72–80. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.2608

[2] "GCC Optimization flags." [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Optimize-Options.html

[3] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics," in *14th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2005, pp. 123–132.

[4] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '04. ACM, 2004, pp. 231–239.

[5] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Exhaustive Optimization Phase Order Space Exploration," in *Fourth Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, New York City, NY, March 2006, pp. 306–318.

[6] J. Gauci and K. O. Stanley, "Autonomous Evolution of Topographic Regularities in Artificial Neural Networks," *Neural Computation*, vol. 22, no. 7, pp. 1860–1898, 2010.

[7] X. Li, M. J. Garzaran, and D. Padua, "Optimizing sorting with genetic algorithms," in *Code Generation and Optimization, 2005. CGO 2005. International Symposium on.* IEEE, 2005, pp. 99–110.

[8] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, pp. 232–275, February 2005.

[9] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[10] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels," *Journal of Physics Conference Series*, vol. 16, pp. 521–530, Jan. 2005.

[11] A. Epshteyn, M. J. Garzarán, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali, "Analytic models and empirical search: A hybrid approach to code optimization," in *Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 259–273.

[12] X. Li, M. J. Garzarán, and D. Padua, "Optimizing Sorting with Genetic Algorithms," in *In Proc. of the International Symposium on Code Generation and Optimization (CGO)*, March 2005, pp. 99–110.

[13] S.-C. Han, F. Franchetti, and M. Püschel, "Program Generation for the All-pairs Shortest Path Problem," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM Press, 2006, pp. 222–232.

[14] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A Case for Machine Learning to Optimize Multicore Performance," *First USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, 2009.

[15] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta Optimization: Improving Compiler Heuristics with Machine Learning," in *Proc. of Programing Language Design and Implementation*, June 2003.

[16] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast Searches for Effective Optimization Phase Sequences," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM Press, 2004, pp. 171–182.

[17] M. Stephenson and S. Amarasinghe, "Predicting Unroll Factors Using Supervised Classification," in *CGO '05: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 123–134.

[18] J. Cavazos and M. 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*. New York, NY, USA: ACM Press, 2006, pp. 229–240.

[19] K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "ACME: adaptive compilation made efficient," in *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, vol. 40. ACM, 2005, pp. 69–77.

[20] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms," *J. Supercomputing*, vol. 36, no. 2, pp. 135–151, 2006.

[21] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A Comparison of Empirical and Model-driven Optimization," in *Proc. of Programing Language Design and Implementation*, June 2003, pp. 63–76.

[22] A. Monsifrot, F. Bodin, and R. Quiniou, "A Machine Learning Approach to Automatic Production of Compiler Heuristics," in *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. London, UK: Springer-Verlag, 2002, pp. 41–50.

[23] 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*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 295–305.

[24] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2009, pp. 75–84.

[25] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle, "MILEPOST GCC: machine learning based research compiler," in *Proceedings of the GCC Developers' Summit*, June 2008.

[26] K. Cooper, P. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*. ACM, 1999, pp. 1–9.

[27] K. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, 2001.

[28] M. R. Jantz and P. A. Kulkarni, "Eliminating false phase interactions to reduce optimization phase order search space," in *CASES*. ACM, 2010, pp. 187–196.

[29] P. A. Kulkarni, D. B. Whalley, and G. S. Tyson, "Evaluating Heuristic Optimization Phase Order Search Algorithms," in *CGO*. IEEE Computer Society, 2007, pp. 157–169.

[30] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Practical exhaustive optimization phase order exploration and evaluation," *TACO*, vol. 6, no. 1, 2009.

[31] E. V. B. Felix V. Agakov and J. C. et al. et al., "Using Machine Learning to Focus Iterative Optimization," in *CGO*, 2006, pp. 295–305.

[32] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Conf. Computing Frontiers*. ACM, 2007, pp. 131–142.

[33] M. Zhao, B. R. Childers, and M. L. Soffa, "A Model-Based Framework: An Approach for Profit-Driven Optimization," in *Proceedings of the the International Symposium on Code Generation and Optimization (CGO)*, 2005, pp. 317–327.

[34] K. Cooper, T. Harvey, and T. Waterman, "An Adaptive Strategy for Inline Substitution," in *Compiler Construction*, ser. Lecture Notes in Computer Science, L. Hendren, Ed. Springer Berlin / Heidelberg, 2008, vol. 4959, pp. 69–84, 10.1007/978-3-540-78791-4_5.

[35] K. D. Cooper, M. W. Hall, and L. Torczon, "An experiment with inline substitution."

[36] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney, "A Comparative Study of Static and Profile-Based Heuristics for Inlining," in *2000 ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*, Boston, MA, Jan. 2000.

[37] K. Hazelwood and D. Grove, "Adaptive Online Context-Sensitive Inlining," in *First Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, San Francisco, CA, March 2003, pp. 253–264. [Online]. Available: http://citeseer.ist.psu.edu/hazelwood03adaptive.html

[38] J. Dean and C. Chambers, "Towards Better Inlining Decisions Using Inlining Trials," in *LISP and Functional Programming*, 1994, pp. 273–282. [Online]. Available: http://citeseer.ist.psu.edu/195684.html

[39] J. Cavazos and M. F. P. O'Boyle, "Automatic Tuning of Inlining Heuristics," in *IN ACM/IEEE CONFERENCE ON SUPERCOMPUTING*, 2005, p. 14.

[40] R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," in *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 1999, pp. 253–256.

[41] K. O. Stanley and R. Miikkulainen, "Efficient Reinforcement Learning Through Evolving Neural Network Topologies," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*. San Francisco: Morgan Kaufmann, 2002, p. 9. [Online]. Available: http://nn.cs.utexas.edu/?stanley:gecco02b

[42] J. A. Alexander and M. Mozer, "Template-Based Algorithms for Connectionist Rule Extraction." in *NIPS*, G. Tesauro, D. S. Touretzky, and T. K. Leen, Eds. MIT Press, 1994, pp. 609–616. [Online]. Available: http://dblp.uni-trier.de/db/conf/nips/nips1994.html#AlexanderM94

[43] H. Lu, R. Setiono, and H. Liu, "Effective data mining using neural networks," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 8, no. 6, pp. 957–961, 1996.

[44] M. W. Craven and J. W. Shavlik, "Extracting tree-structured representations of trained networks," *Advances in neural information processing systems*, pp. 24–30, 1996.

[45] ——, "Learning Symbolic Rules Using Artificial Neural Networks," in *Proceedings of the Tenth International Conference on Machine Learning*. Morgan Kaufmann, 1993, pp. 73–80.

[46] L. A. Smith, J. M. Bull, and J. Obdrzálek, "A Parallel Java Grande Benchmark Suite," in *SC2001: High Performance Networking and Computing. Denver, CO, November 10–16, 2001*, ACM, Ed. ACM Press and IEEE Computer Society Press, 2001.

[47] C. Liao, D. J. Quinlan, R. W. Vuduc, and T. Panas, "Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization." in *LCPC'09*, 2009, pp. 308–322.

[48] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Virtual Machine," *IBM Systems Jounrnal*, vol. 39, no. 1, 2000.

[49] Website, "Java Grande Benchmarks." [Online]. Available: http://www2.epcc.ed.ac.uk/computing;http://sequential.html

[50] T. Li, L. K. John, N. Vijaykrishnan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy, "Using complete system simulation to characterize SPECjvm98 benchmarks," in *ICS*, 2000, pp. 22–33.

[51] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "SPECjvm2008 Performance Characterization," in *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 17–35. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-93799-9_2

[52] "Dacapo Benchmark Suite." [Online]. Available: http://dacapobench.org/benchmarks.html

[53] K. O. Stanley, "Efficient evolution of neural networks through complexification," Ph.D. dissertation, The University of Texas at Austin, 2004, supervisor-Miikkulainen, Risto P.

[54] X. Yao, "Evolving Artificial Neural Networks," 1999.

[55] K. O. Stanley and R. Miikkulainen, "Evolving Neural Network through Augmenting Topologies." *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: http://dblp.uni-trier.de/db/journals/ec/ec10.html#StanleyM02

[56] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[57] L. A. Smith, J. M. Bull, and J. Obdrzálek, "A Parallel Java Grande Benchmark Suite," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 2001, pp. 8–8. [Online]. Available: http://portal.acm.org/citation.cfm?id=582034.582042;http://www.bibsonomy.org/bibtex/2c43a9020f22de7cdb6608093a3074ea6/gron

[58] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis."

[59] S. M. Blackburn *et al.*, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *21st Object-Oriented Prog. Syst., Lang., & Appl. Conf. Proc. (OOPSLA)*, Oct. 2006, pp. 169–190.

[60] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder, "Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine." in *OOPSLA 2011*, C. V. Lopes and K. Fisher, Eds. ACM, 2011, pp. 657–676. [Online]. Available: http://dblp.uni-trier.de/db/conf/oopsla/oopsla2011.html#SeweMSB11;http:

//doi.acm.org/10.1145/2048066.2048118;http://www.bibsonomy.org/bibtex/23b9b03b5a29dc0a46fa0f4fd7e6b96f6/dblp

[61] S. Luke, "A Java-based Evolutionary Computation Research System," ECJ 11: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2004.