Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning

Sameer Kulkarni John Cavazos
University of Delaware
{skulkarn,cavazos}@cis.udel.edu

Abstract

Today's compilers have a plethora of optimizations to choose from, and the correct choice of optimizations can have a significant impact on the performance of the code being optimized. Furthermore, choosing the correct order in which to apply those optimizations has been a long standing problem in compilation research. Each of these optimizations interacts with the code and in turn with all other optimizations in complicated ways. Traditional compilers typically apply the same set of optimization in a fixed order to all functions in a program, without regard the code being optimized.

Understanding the interactions of optimizations is very important in determining a good solution to the phase-ordering problem. This paper develops a new approach that automatically selects good optimization orderings on a per method basis within a dynamic compiler. Our approach formulates the phase-ordering problem as a Markov process and uses a characterization of the current state of the code being optimized to creating a better solution to the phase ordering problem. Our technique uses *neuro-evolution* to construct an artificial neural network that is capable of predicting beneficial optimization ordering for a piece of code that is being optimized. We implemented our technique in Jikes RVM and achieved significant improvements on a set of standard Java benchmarks over a well-engineered fixed order.

General Terms Performance, Compilation, Compiler Optimization

Keywords Phase Ordering, Compiler optimization, Machine learning, Neural Networks, Java, Jikes RVM, Code Feature Generation

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Selecting the best ordering of compiler optimizations for a program has been an open problem in compilation research for decades. 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 developers of GCC 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 phases in a fixed order on each method of a program. However, several researchers [2, 3, 15], 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 paper 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 Java JIT compiler and automatically determine good phase-orderings of optimizations on a *per method* basis. Rather than developing a handcrafted 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 induced using *Neuro-Evolution for Augmenting Topologies* (NEAT) [23].

A trained ANN uses input properties (i.e., features) of each method to represent the current optimized state of the method and given this input, the ANN outputs 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

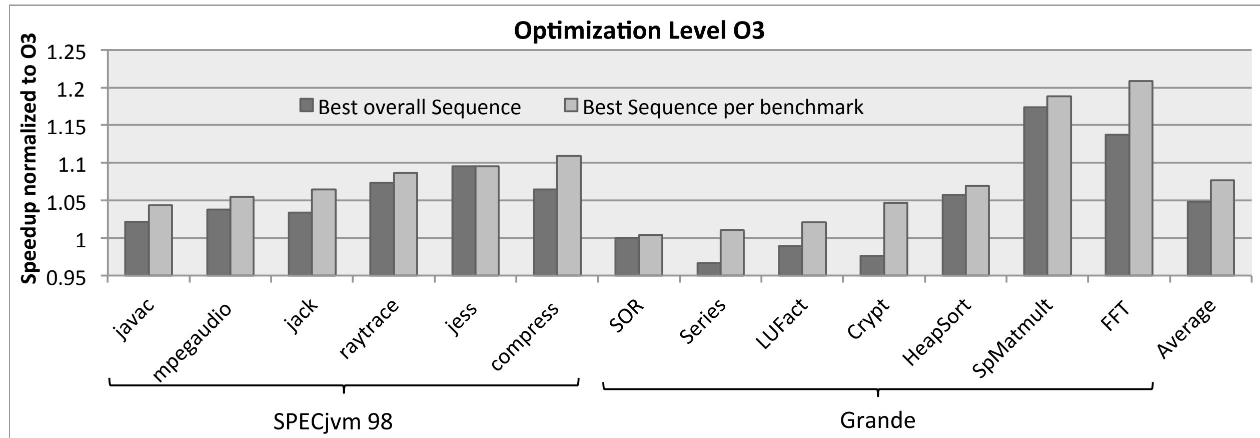

Figure 1. We used genetic algorithms to find "tuned" optimization sequence for benchmarks in the SPECjvm98 and Java Grande benchmark suites. In a first experiment, we obtained an optimization ordering that performed well over all the benchmarks by using the running time of all the benchmarks as a fitness function. In a second experiment, we used running time of each individual benchmark to "evolve" the best optimization ordering for each benchmark.

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. This 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 3.

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 in this paper is the first to automatically induce a heuristic that can predict an overall optimization ordering to 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.[1] Furthermore, we show significant performance improvement over an existing well-engineered compilation system. We make the following contributions:

- We present a method of evolving an ANN to be used for phase-ordering, which, to the best of our knowledge, is the first technique that "learns" from characteristics of code being optimized what is the best ordering of optimizations to apply.
- We show that our phase-ordering technique can achieve significant speedup over the traditional approach of applying a fixed optimization sequence.
- We compare our ANN-based phase-ordering technique to the current state-of-the-art phase-ordering technique, i.e., using genetic algorithms (GAs). Moreover, we show that our technique is much more practical for phase-ordering than using GAs.
- We present results optimizing several Java benchmark programs to illustrate that optimization order is important.
- We show that our trained ANN generates a customized phase-ordering for a variety of different methods in various SPECjvm2008 and DaCapo benchmarks to create a truly method-specific phase-ordering compiler.

2. Overview

Compiler optimizations transform the code being optimized, thus the application of each optimization potentially affects the benefit of downstream optimizations. One of the most prominent examples of this is the phase-ordering problem between register allocation and instruction scheduling. However, any set of optimizations can potentially interact with each other and can therefore participate in a phase-ordering problem. 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.

2.1 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 [3,

[1] The same technique can be applied in a static compilation setting in a straight-forward manner.

6–9, 14, 16, 16, 17], to show that selecting the best ordering of optimizations has the potential to significantly improve the running time of dynamically-compiled programs.

We used GAs to construct a custom ordering of optimizations for each of the Java Grande [22] and SPEC JVM 98 benchmarks.[2] In this GA approach, 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 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.

We ran two different experiments using GAs. The first experiment consisted of finding the best optimization sequence across our benchmarks. Thus, we evaluated 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. That is, we used average speedup of our benchmarks (normalized to opt level O3) as our fitness function for each chromosome. This result corresponds to the "Best Overall Sequence" bars in Figure 1. The purpose of this experiment was to discover the optimization ordering that worked best on average for all our benchmarks.

The second experiment consisted of finding the best optimization ordering for each benchmark. Here, 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 1. This represents the performance that we can get by customizing an optimization ordering for each benchmark individually.

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. In contrast, 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.

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

2.2 Issues with Current State-of-the-Art

Using genetic algorithms is the current state-of-the-art in obtaining good optimization orderings, and they can bring significant performance improvements for some programs. However, using GAs has several issues that impede their widespread adoption in traditional compilers.

Expensive Search: GAs and other search techniques are inherently expensive because they need to 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 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. We show empirical results in Section 7 describing the time it took to for GAs to construct optimization orderings.

Method-specific difficulty: Using GAs to find custom orderings of optimizations for specific code segments within 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. [3]

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 that would have to be evaluated 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.

2.3 Proposed Solution

Instead of using expensive search techniques to solve the phase-ordering problem, we 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.

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

[3] Evaluating optimization orders for a method outside of an application context [19] 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.

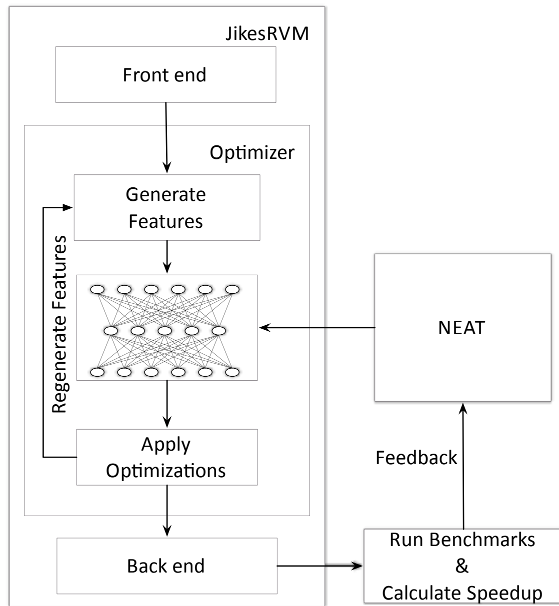

Figure 2. The figure above represents the framework used to evolve a neural network using NEAT to guide the compilation of a given method. The Figure 4 describes the way the neural network was used to guide the compilation process.

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.

In this paper we choose 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 2 and described in detail in Section 3. 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 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.

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. NEAT performs neuro-evolution to construct a neural network to be used for phase-ordering. The above steps describe the process of evaluating an ANN as part of neuro-evolution.

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, and if the platform were to change, the entire tuning process of the heuristic would have to be repeated.

3. Approach

This section gives a detailed overview of how neuro-evolution based machine learning is used to construct a good optimization phase-ordering heuristic for the optimizer within Jikes RVM. The first section outlines the different activities that take place when training and deploying a phase-ordering heuristic. This is followed by sections 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 outlines our technique.

3.1 Overview of Training and Deployment

There are 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.

As part of 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 a feature vector) 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 high-

est 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.

Once the best ANN is evolved, it is installed into the Jikes RVM compiler and used at runtime as an optimization heuristic. The next sections describe these stages in more detail.

3.2 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. In this paper, we propose 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. 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.

3.3 Neuro-Evolution Overview

In the this paper, we use 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 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 3.5.3. Once the initial set of generated neural networks are evaluated, the 10 best neural networks from this set are propagated to the next generation and are also used to produce new neural networks in this generation.

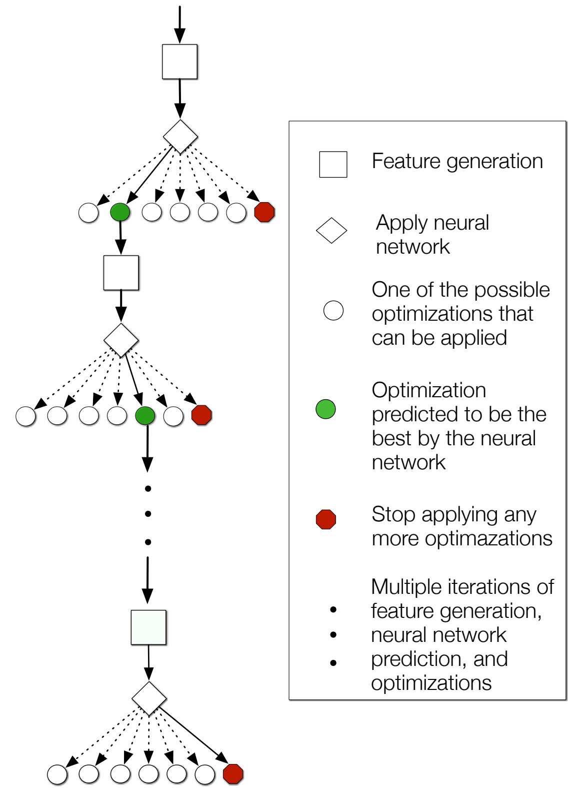

Figure 4. This figure represents the phase-ordering process. The process starts when the Jikes RVM optimizer receives a method to optimize. We iterate over the instructions of the method to generate the features, and then provide these features to the neural network. The neural network then provides a set of outputs, which represent the probabilities of each optimization being beneficial. The optimization with the highest probability is applied to the code. One of the outputs of the network corresponds to "stop optimizing." When the probability of this output is highest, the optimizer stops applying optimizations to the method.

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 probabilties are within the ranges suggested by the authors of NEAT. Neurons are re-

Feature	Meaning
bytecodes	Number of bytecodes 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 bytecodes that ...
aload, astore	are Array Loads and Stores
primitive, long	are Primitive or Long computations (e.g., iadd, fadd)
compare	are Compares (e.g., lcmp, dcmpl)
branch	are Branches (forward/backward/cond/uncond)
jsr	are a JSR
switch	are a SWITCH
put/get	are a PUT or GET
invoke	are an INVOKE
new	are a NEW
arraylength	are 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 1. Method features being collected. To reduce the length of the table several (different) features have been placed in logical groups.

moved 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 2 depicts the process of constructing a neural network using NEAT to replace the optimization heuristic in Jikes RVM.

3.4 Feature Extraction

Determining the properties of a method that predict an optimization improvement is a difficult task. As we are operating in a dynamic compilation environment, 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 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 bytecodes belonging to a particular category (e.g., 30% loads, 22% floating point, 5% yield points, etc.).

3.5 Applying NEAT

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

3.5.1 Why NEAT?

NEAT can be used to solve challenging tasks because it can evolve networks of unbounded complexity from a minimal starting point. This method has been shown to outperform the best fixed-topology method on challenging reinforcement learning tasks [23] The reason that NEAT is faster and better than typical reinforcement learning is three-fold: 1) it incrementally grows networks from a minimal structure, 2) it protects structural innovation using natural selection, and 3) it employs a principled method of crossover of different topologies.

Neural networks are traditionally trained using supervised learning algorithms, which require a labeled training set. The 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 the 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 2.3.

3.5.2 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 numbers 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 4 shows the process of phase-ordering.

3.5.3 Fitness Functions

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 benchmark 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.

4. Infrastructure + Methodology

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

4.1 Platform

For our experiments in this paper, we modified version 3.1.1 of the Jikes Research Virtual Machine [4]. 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.

4.2 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 [29]. These benchmarks were used for training primarily due to their short execution times.

For the test set, we used the SPECjvm98 [28], the SPECjvm2008 [27], and the DaCapo benchmark [1] 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. [4] 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 5 come from the benchmarks in our test set.

[4] 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 2. The set of optimizations that were used to perform phase ordering in our experiments.

4.3 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 loaded methods at the highest optimization level. Under the adaptive scenario, all dynamically loaded methods are first compiled by the baseline compiler that converts bytecodes 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.

4.4 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 bytecodes 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 durations, 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 bytecodes 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 in this paper was less that 1.2% and the average noise was 0.7%.

4.5 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.

5. Results

In this section, we present our results of using the neural network that performed best on the training set. We used this network to determine good optimization orders for methods in programs from the SPECjvm98, SPECjvm2008, and DaCapo benchmark suites in both an adaptive and optimizing compilation scenario.

5.1 Adaptive Compiler

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 4.3. 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%.

5.1.1 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.

5.1.2 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.

5.2 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%.

5.2.1 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.

Total time When measuring total time, we observed a modest increase in performance of around 3.4%. The best

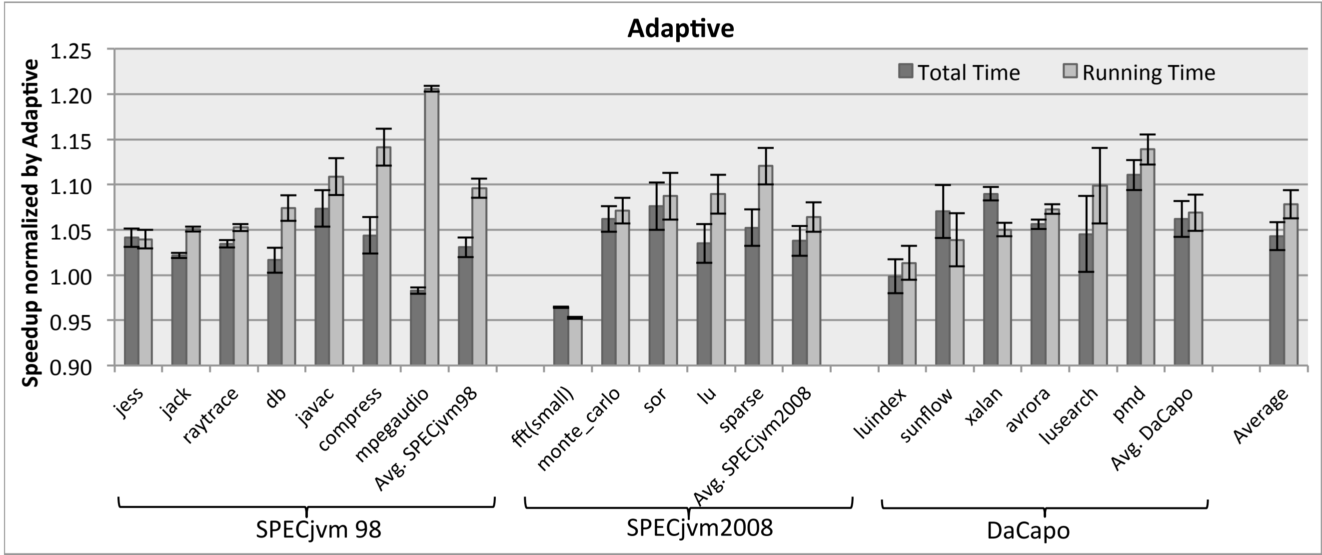

Figure 5. Adaptive: The graph above represents the speedup achieved by using NEAT when used by Jikes RVM in adaptive mode to optimize each benchmark in the test set. We compare our result with the performance of each of the benchmarks when using the default adaptive compilation scenario.

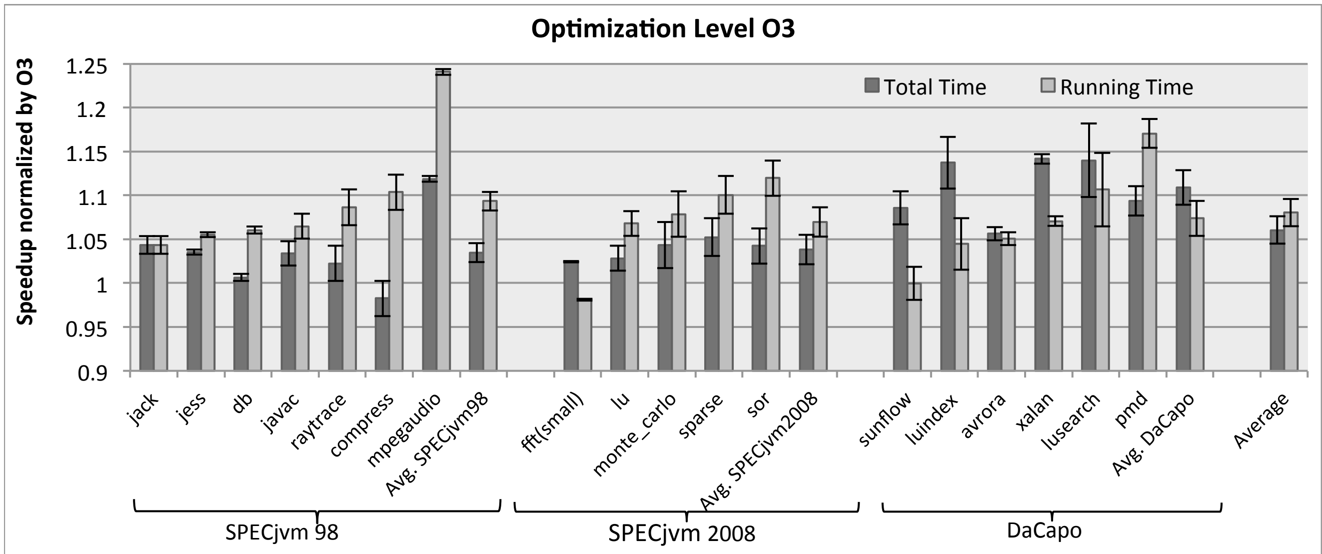

Figure 6. Optimization Level O3: The graph above represents the speedup achieved by using NEAT when used in the non-adaptive mode in Jikes RVM to optimize each benchmark in the test set. We compare our result with the performance of each of the benchmarks when using the default non-adaptive compilation scenario.

performing benchmark was again `mpegaudio` at 11% speedup.

5.2.2 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

```
factor()
{
  for (...) {
    arithematic operation over an array
    for (...) {
      arithematic operation over an array
    }
  }
  ...
  if(...){
    for (...) {
      arithematic operation over an array
    }
  }
  if(...){
    for (...) {
      for (...) {
        arithematic operation over an array
      }
    }
  }

}
```

Listing 1. Pseudo-code for scimark.lu.LU.factor, the hottest method for the SPEC2008 lu benchmark

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

Listing 2. Slow Code (SA applied before BRO): The following code corresponds to a *while loop*, where *n* iterations of the loop require *n* conditional jumps and *n* unconditional jumps. This is the code produced by using optimization level O3.

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

Listing 3. Fast Code (BRO applied before SA): The following code corresponds to a *do-while loop*, where *n* iterations of the loop would require *n+1* conditional jumps but no unconditional jumps, this improves the performance. This is the code produced using the optimization ordering produced using our neural network.

Figure 7. Listing 1 shows the pseudo code of the scimark.lu.LU.factor method that is compiled by the optimizing compiler. The two HIRs generated for scimark.lu.LU.factor by the two different optimization orderings are shown in Listing 2 and Listing 3. Changing the order that the transformations are applied changes the running time by almost 7%

length helped to reduce the amount of compilation required, and thus improves total time performance.

5.2.3 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%.

6. Exploration of Phase ordering benefit

In this section, we tried 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 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 4 is the hottest method in the scimark.lu.small benchmark. The figure also shows code after applying *Branch Optimization* before *CFG Structural Analysis* (i.e., the ordering obtained from the default optimization level) and the code 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. A snippet of representative code is shown in Figure 7. 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 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.

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 Figure 8. Considering the number of nested loops in this method,*Loop Unrolling* could potentially be an optimization to this method. Howere we realized that our neural network applied *CFG*

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

Listing 4. Pseudo-code for matmult, the hottest method for the SPEC2008 sparse benchmark.

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

Listing 5. Slow Code (LU applied before SA): Here the number. of unconditional statements are unnecessary, and hampers the performance of the code. This is the code produced by using optimization level O3.

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

Listing 6. Fast Code (SA applied before LU): During compilation, *CFG Structural Analysis* was applied before *Loop Unrolling*, which gave the compiler a chance to clean up the code before the loop unrolling was applied. This is the code produced using the optimization ordering produced using our neural network.

Figure 8. The final HIR generated for sparse.SparseCompRow.matmult by the two different optimization orderings. The code generated by applying *CFG Structural Analysis* before *Loop Unrolling* shown in Listing 6 performs better in terms of running time and achieved a speedup of almost 14%.. When looking at the other characteristics, the number of unconditional jumps were reduced by 33% and there was a 10% reduction in the number of basic blocks.

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 differences in the machine code that were generated. 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 optimum. Figure 8 shows a piece of machine code that is less than otimal. When looking at this particular instance we quickly relaized 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.

7. Training Time

Training our machine learning heuristic requires us to provide fitness values to each of the heuristics being tested. In our neuro-evolution scenario, the fitness of the heuristic can only be measured in terms of the performance of the benchmark when this heuristic is applied.

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 5. Given the number of days that it can take to train each benchmark we feel that is is impractical to use GA's for phase-ordering, especially within a dynamic compilation scenario.

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 4 and Figures 5 and 6. This network had 30 inputs, 14 outputs, 24 hidden nodes, and 503 total connections.

Reduction of optimization sequence length From our experiments, we were able to demonstrate two achievements.

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
SPECjvm2008 contd.	
sparse	20
sor	16
DaCapo	
avrora	19
luindex	16
lusearch	16
pmd	18
sunflow	16
xalan	17
Average	**17**
Default	**23**

Table 3. The average number of optimizations that were applied by the neural network.

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 4. 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 4, 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 9. Figure 9 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

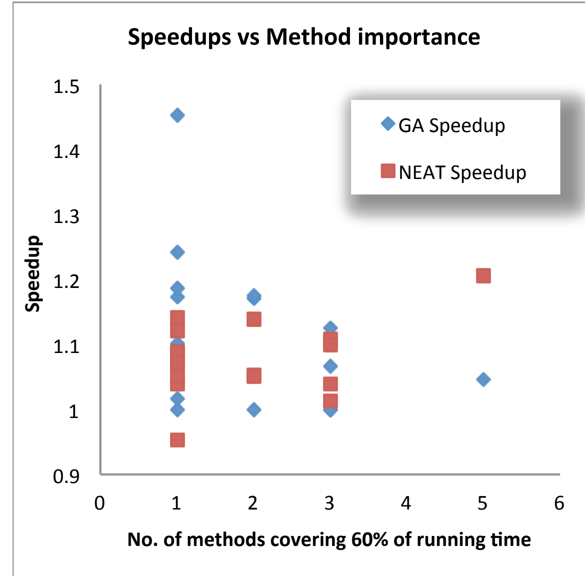

Figure 10. Speedup based on method importance: The plot above represents the speedup achieved by evolving an optimization sequence using genetic algorithm per benchmark and NEAT. Each data point in the plot corresponds to a benchmark, and the plot depicts the number of methods that constitute 60% of the running time for a particular benchmark versus the speedup obtained for that benchmark.

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 For some benchmarks, 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. 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.

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. This was the method proposed by Cooper *et al.* [6] and was shown to find good optimization sequences for a program. Figure 10 shows the speedup achieved by both GAs and neural networks on each benchmark as it relates to the

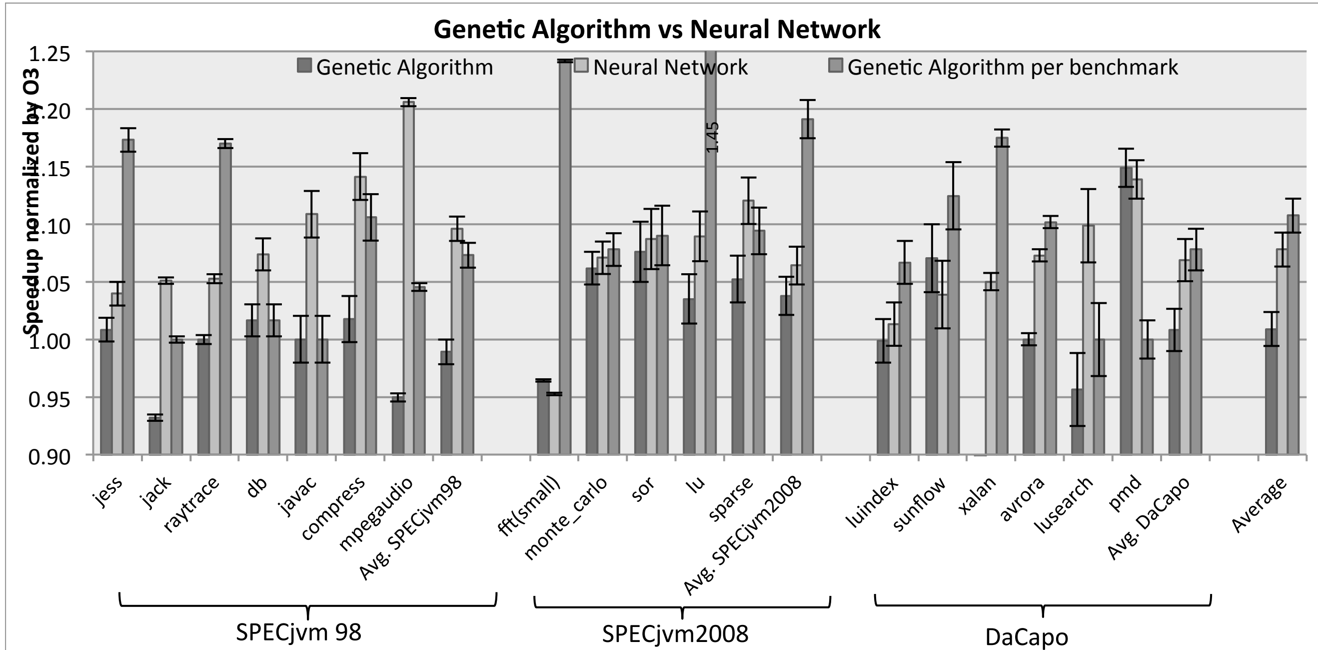

Figure 9. Genetic Algorithm versus Neural Network: The graph above represents the speedup achieved by the best optimization sequence found by the genetic algorithm for all the benchmarks in the training set, when applied to the test set (JikesRVM in non-adaptive mode). We compare our result with the performance of each of the benchmarks when using the default non-adaptive compilation scenario.

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 4. This table gives information about the hottest methods in SPECjvm2008 and the optimization sequences obtained from our neural network for each of these methods. The abbreviations used to described the sequences are explained in Table 2.

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 9, 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.

9. Related Work

Auto-tuning: An area that is closely related to this paper is the study of automatic code generation and optimization for different computer architectures (auto-tuning), which has been explored in many existing studies for many different applications. A number of library generators automatically produce high-performance kernel routines [21, 26, 30]. Recent research efforts [12, 18] 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,

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 5. This table shows the average time that we have taken if we evolved an optimization ordering using GA for each benchmark individually.

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

Table 6. Time taken in days to train the training set, to provide the reults in Figure 9

Ganapathi [11] *et al.* 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. This 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.

Machine learning applied to Compilation: Machine learning and search techniques applied to compilation has been studied in many recent projects [5, 8, 9, 14, 20, 24, 25, 31]. 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 in this paper is motivated by these studies and makes a significant step forward: the compiler will not only use program characteristics, but will also learning to decide the right ordering of optimizations.

Several researchers have looked at searching for the best sequence of optimizations for a particular program [6–8, 13, 16, 17], for example the work by Cooper *et al.* [6] 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 *et al.* [8] propose 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 has 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.* [17] 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, the techniques in this paper characterized methods being optimized; therefore, the techniques described here learn which optimizations are beneficial to apply to "unseen" methods with similar characteristics.

Many researchers have also looked at using machine learning to construct heuristics that control compiler optimizations. Cavazos *et al.* [5] used logistic regression to control what optimizations to apply in JikesRVM. However, they do 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 likely because 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.* [25] 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.* [20] 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.* [2] 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 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 2.1. 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.

Fursin *et al.* [10] (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".

10. Conclusion

This paper has shown that method-specific optimization orderings can give significant performance improvements within the Jikes RVM JIT compiler. It has also demonstrated that a technique of neuro-evolution can automatically derive a neural network that gives significant performance improvements over a well-engineered optimization ordering. We show total execution time improvements of up to 20%. To the best of our knowledge, this is the first paper to demonstrate that machine-learning models can be successfully used to choose optimization orders for methods within a compiler. The present study is promising as it provides a fresh prospective to the problem of phase ordering which has been studied for decades. The amount of improvement that can be found from this method has not yet reached its full potential, and we propose to improve the machine learning algorithm to provide better improvements.

For future work, we would like to implement similar phase-ordering techniques in a static compiler, in order to understand the behavior of other environments on our setup. There is nothing about the technique that makes it specific to dynamic compilation. In addition, we would also like to incorporate profile information into the feature set, which allow us to improve our predictions.

References

[1] Dacapo benchmark suite. URL `http://dacapobench.org/benchmarks.html`.

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

[3] 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*, LCTES '04, pages 231–239. ACM, 2004.

[4] 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*, 39(1), 2000.

[5] 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*, pages 229–240, New York, NY, USA, 2006. ACM Press.

[6] 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*, pages 1–9. ACM, 1999.

[7] K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2001.

[8] 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*, volume 40, pages 69–77. ACM, 2005.

[9] 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*, 36(2):135–151, 2006.

[10] 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.

[11] 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.

[12] 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*, pages 222–232, New York, NY, USA, 2006. ACM Press.

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

[14] 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*, pages 171–182. ACM Press, 2004.

[15] 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*, pages 306–318, New York City, NY, March 2006.

[16] P. A. Kulkarni, D. B. Whalley, and G. S. Tyson. Evaluating heuristic optimization phase order search algorithms. In *CGO*, pages 157–169. IEEE Computer Society, 2007.

[17] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *TACO*, 6(1), 2009.

[18] 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)*, pages 99–110, March 2005.

[19] 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*, pages 308–322, 2009.

[20] 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*, pages 41–50, London, UK, 2002. Springer-Verlag.

[21] 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*, 93(2):232–275, February 2005.

[22] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In ACM, editor, *SC2001: High Performance Networking and Computing. Denver, CO, November 10–16, 2001*. ACM Press and IEEE Computer Society Press, 2001. ISBN 1-58113-293-X.

[23] 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)*, page 9, San Francisco, 2002. Morgan Kaufmann. URL `http://nn.cs.utexas.edu/?stanley:gecco02b`.

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

[25] 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.

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

[27] Website. Specjvm 2008, . URL `http://www.spec.org/jvm2008/`.

[28] Website. Specjvm 98, . URL `http://www.spec.org/jvm98/`.

[29] Website. Java grande benchmarks, . URL `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/sequential.html`.

[30] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[31] 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*, pages 63–76, June 2003.