# Automatic Construction of Inlining Heuristics using Machine Learning

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

Christian Wimmer Douglas Simon
Oracle Labs
{christian.wimmer,doug.simon}@oracle.com

## Abstract

Method inlining is considered to be one of the most important optimizations in a compiler. However, a poor inlining heuristic can lead to significant degradation of a program's running time. Therefore, it is important that an inliner has an effective heuristic that controls whether a method is inlined or not. An important component of any inlining heuristic are the features that characterize the inlining decision. These features often correspond to the caller method and the callee methods. However, it is not always apparent what the most important features are for this problem or the relative importance of these features. Compiler writers developing inlining heuristics may exclude critical information that can be obtained during each inlining decision.

In this paper, we use a machine learning technique, namely neuro-evolution [18], to automatically induce effective inlining heuristics from a set of features deemed to be useful for inlining. Our learning technique is able to induce novel heuristics that significantly out-perform manually-constructed inlining heuristics. We evaluate the heuristic constructed by our neuro-evolutionary technique within the highly tuned Java HotSpot server compiler and the Maxine VM C1X compiler, and we are able to obtain speedups of up to 89% and 114%, respectively. In addition, we obtain an average speedup of almost 9% and 11% for the Java HotSpot VM and Maxine VM, respectively. However, the output of neuro-evolution, a neural network, is not human readable. We show how to construct more concise and readable heuristics in the form of decision trees that perform as well as our neuro-evolutionary approach.

*CGO'13* 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

## 1. Introduction

Method inlining can substantially alter the execution time of the code being compiled. It eliminates the overhead of a call and broadens the scope for other optimizations. Careless use of method inlining however can increase register pressure and can dramatically increase the memory footprint of a program, thus degrading the program's run time. In addition, it can make the compiler perform extra work with little or no benefit, which also impacts a program's execution time in a dynamic compilation scenario. Therefore, inlining heuristics must be carefully tuned to enhance the benefits of inlining while avoiding its potential costs. Determining whether or not to inline a method is a difficult decision and depends on a number of factors, including the calling context of the method, the target platform, and other optimizations that may be applied downstream during the compilation process.

As with many compiler optimizations, a heuristic typically controls when and to what extent inlining should be applied. These heuristics are carefully tuned by compiler experts; a difficult and time-consuming task. Also, the heuristics are inherently static, hard-coded into the compiler and often do not take into account the changing environment or platform. In the case of the Java HotSpot server compiler [14] and the Maxine VM C1X compiler [20], the compilers considered in this paper, the same inline heuristic is used for all the platforms they support.

A good solution to any inlining decision also requires considering the effect of previous inlining decisions. That is, a callee is a good candidate for inlining depends on how many other methods have been previously inlined into the caller. Also, if a callee method is called at two different locations in the caller method, each callee should be considered as a distinct decision point. Thus, inlining is a difficult optimization for which to construct heuristics manually.

In addition, inlining typically has a fairly large set of decision points, thus creating a large space of possible solutions. This makes it impossible to exhaustively iterate over the complete search space to find an optimal solution. For these reasons machine learning is a prime candidate for automatically constructing good inlining heuristics.

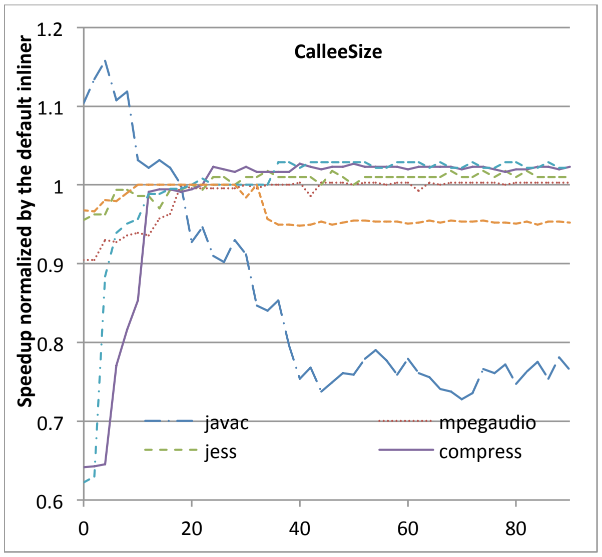

**Figure 1.** Behavior of SPECjvm98 benchmarks on the Maxine VM when varying the maximum size of the callee that is eligible for inlining. Notice that there is not one value that can provide the best performance over all the benchmarks.

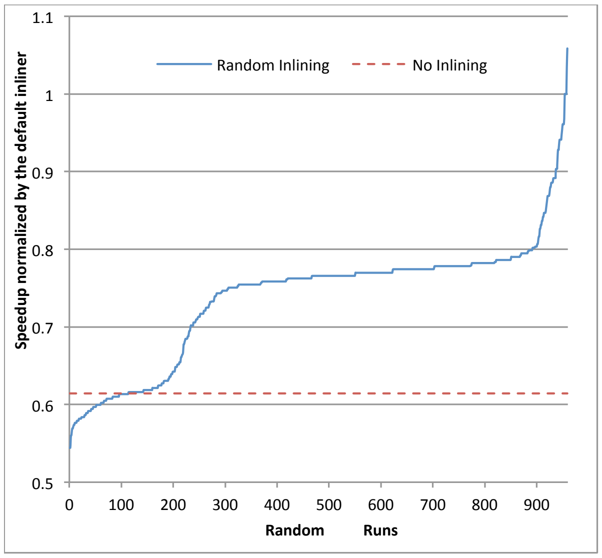

**Figure 2.** Performance achieved by the `raytrace` benchmark of the SPECjvm98 benchmark suite when on the Maxine VM when using a random inliner. This inliner makes a random choice of whether to inline or not at every inlining decision point.

The traditional method of using machine learning to generate heuristics is through supervised learning. Supervised learning requires a heuristic be developed over a training set, where each decision point is labelled with the correct decision. In the case of method inlining, a decision point would comprise a single method call consisting of a caller and a callee pair. In method inlining, each decision can affect subsequent decisions, thus making it difficult to study the effect of a single method inlining decision independent of the other decisions. Therefore, it is more appropriate to use unsupervised learning in this situation. In unsupervised learning, we generate solutions to the problem and use this solution to measure its effectiveness. The measure of the solutions' effectiveness is then used as feedback to generate subsequent prototype solutions, thus overcoming the absence of a labelled dataset. In this paper, we use an unsupervised learning method called Neuro-Evolution of Augmenting Topologies (NEAT) to construct an effective heuristic to perform inlining. However, the heuristic generated by this technique is unreadable by humans. We thus use our NEAT heuristic to construct a labelled data set which we then use to construct a more readable heuristic using decision trees.

In summary, the new contributions of this paper are:

- We show that using an automated method of building inlining heuristics requires no human intervention, can find new features that are important in making inlining decisions, and can generate heuristics that are as good, if not better, than state-of-the-art for code.
- We show how to automatically construct effective inlining heuristics using an unsupervised machine learning algorithm called NEAT.
- We present a human-readable heuristics with decision trees using the heuristics NEAT produces.
- We compare our different approaches to constructing inlining heuristics to the state-of-the-art for producing effective inlining heuristics.
- We show that in some cases this heuristic works much better than the default inlininer in compiling code written in Scala.

## 2. Motivation

The growing popularity of managed programming languages makes it crucial to construct better compilers that can translate programs into the most efficient machine code. Also, compilers have to be re-tuned for each new platform or when certain changes are made to the compiler, e.g., when new compiler optimizations are added. Developing and tuning compiler optimizations in this changing environment becomes difficult. This is especially true for method inlining, which is one of the most important optimizations in a compiler. This is because the heuristics that control inlining are composed of various metrics that must all be carefully considered. In this section, we demonstrate the sensitivity of certain metrics to the inlining decision.

***Sensitivity of Heuristic Metrics.*** Inliners use a variety of different metrics to make informed decisions. In traditional compilers, each metric is often compared to a carefully selected threshold that guides the inlining decisions to be made. However, we have found that a set of thresholds that works best for one benchmark may not work as well for another benchmark.

Figure 1 shows the sensitivity of the *callee size* metric in the Maxine VM for a variety of different Java benchmarks. The callee size is the length of the callee method's bytecode array. The threshold chosen for the callee size controls the size of the callee methods inlined into the caller methods. The inliner will potentially continue to inline callee methods as long as their size is under the callee size threshold.

The *x-axis* shows the different values we used for the callee size threshold starting at 0 and going to 100, and the *y-axis* shows the performance of the Java benchmarks relative to using the default inliner. For example a value of .95 means that our heuristic was 5% slower than the default inlining heuristic present in the Maxine VM. This graph shows that each benchmark achieves peak performance at different values of the callee size metric. For example, `javac` achieves close to its best performance when the callee size threshold is 0 meaning that no inlining will occur. However, this is the worst threshold value to use for both `compress` and `raytrace`. Interestingly, the benchmarks do not follow any particular trend based on this metric.

In addition to callee size there are several other metrics that can affect inlining performance. It is difficult to understand the exact relationship between these metrics because these metrics are interdependent. For example, if there are only two metrics to consider, inline depth and the maximum size of the callee, then increasing the inline depth would require the compiler writer to reduce the maximum allowed size of the callee method in order to avoid code bloat.

***Search Space*** Along with the variety and sensitivity of various inlining metrics, any inlining decision has the potential to affect subsequent inlining decisions. Thus the space of possible inlining solutions is large. However, our experiments show that the set of good points within this large search space is small. Consider the example of a random inliner, which at each inlining decision point makes a random choice whether or not to inline. We ran 1000 such trials on all the benchmarks of the SPECjvm98 benchmark suite.

Figure 2 shows 1000 runs of the random inliner inlining the `raytrace` benchmark in the SPECjvm98 benchmarks suite. On the *x-axis* we show the 1000 random trials sorted by the speedup over the default Maxine inliner. We also provide a line at 61% which represents the performance of the benchmark with no inlining. We observed an average slowdown of around 27% for our random inliner over the default inliner. We also saw almost 10% of the cases performed worse than no inlining. In addition, there is a small fraction of space explored that has points that outperformed the default inliner. Clearly search can be used to find good inlining decisions. In fact, Cooper *et al.* [6] use search to find good solutions in the inlining solutions space. As shown above, random searching would eventually give us a good inlining solution, however this is expensive and not practical. This process would have to be repeated for each program, because random inlining does not give us a heuristic that we can use for new programs. We would like a search method that not only finds good points in the search space, but also gives us knowledge that can be reused for subsequent programs.

One such solution involves using Neuro-Evolution of Augmenting Topologies (NEAT). In this method, we generate a random set of neural networks for an initial generation, each of which can be used as a method inlining heuristic. Performance of each neural network is measured on a set of programs called the *training set*, and the best performing neural network is allowed to propagate to the next generation. This process of natural selection picks the best neural network to generate over other neural networks. Eventually over a span of multiple generations we get a heuristic that can be used as a solution not just for the program NEAT trained on, but any other program that it has not seen before.

## 3. Problem and Approach

Most compilers have static rules to guide inlining and are tuned for a set of benchmarks. This tuning process is performed manually and is tedious and relatively brittle. Also, creating the heuristic and tuning it needs to be repeated each time the compiler is modified for a new platform or when a new optimization is added to the compiler. In this paper, we propose to use machine learning as a way to overcome the shortcomings of manually constructing the inlining heuristic. Our approach eliminates the requirement of having a compiler writer actually spend time to construct or even tune the heuristic, and still be able to produce a good and well tuned heuristic.

We also compare different machine learning algorithms and discuss their individual advantages and disadvantages. The different machine learning algorithms that we discuss in this paper are: Neuro-Evolution of Augmenting Topologies (NEAT), Genetic Algorithms (GAs), and Decision Trees (DTs).

NEAT is an unsupervised learning algorithm that constructs heuristics in the form of neural networks to guide the method inlining process. The neural network is given inputs that characterizes an inlining decision point involving a particular caller and a callee method, and the output of the network corresponds to whether the callee should be inlined to the caller. The network is used as the inlining heuristic, and the running time of the program is used as feedback of the effectiveness of the solution. Cavazos *et al.* [5] previously used genetic algorithms (GAs) to construct highly-tuned inlining heuristics, however in this paper, we show that NEAT can out-perform genetic algorithms at the task of developing inlining heuristics. NEAT can generate effective inlining heuristics, but the heuristics are in the form of neural networks, which makes it difficult to discern intuition of how the heuristic is operating. Therefore, in this paper, we show how to develop readable inlining heuristics that perform as well as the NEAT heuristics using decision trees.

```
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 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.** Inlining heuristic of the server compiler

In order to generate a decision tree, a *labelled training set* is required. A labelled training set consists of a set of input features characterizing a specific decision point and a labelled 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 labelled 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 labelled training sets using NEAT heuristics in greater detail in Section 3.4.

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 heuristics constructed by our machine learning approaches are as fast as manually-constructed heuristics. Thus, there is no overhead incurred by using heuristics constructed with machine learning, nor are there any major code changes.

## 3.1 Default Inlining Heuristic

Figure 3 depicts high-level pseudo-code of the default inlining heuristic found in the Maxine VM. This heuristic decides whether or not to inline based on a series of manually-constructed tests. Before this heuristic is invoked, a collection of checks are made to make sure that inlining is safe to perform and does not cause invalid code to be generated, e.g., if the callee method is virtual. The inlining tests involve characteristics of the callee method to be potentially inlined, the inline depth, and the size of the intermediate representation.

The first test checks for small methods (methods smaller than the parameter ALWAYS_INLINE_SIZE). If the method is small further tests are required, but if the method is larger than the ALWAYS_INLINE_SIZE parameter, we perform a second test that restricts methods that are particularly large (greater than CALLEE_MAX_SIZE) from being inlined. The intuition for this test is that large callee methods greater than some threshold size will cause code bloat and reduce the benefit of inlining. The default inlining heuristic performs a third test that imposes a limit on the maximum depth of the inlining decisions at any given call site. The fourth and last test checks whether the estimated size of the caller method is large (greater than CALLER_MAX_GRAPH_SIZE). Caller method is estimated by the variable *currentGraphSize*, which corresponds to the size of the caller's intermediate representation, including the increased nodes added from previous inlining performed. Finally, if all tests are false, the callee method is inlined. In Section 3.5, we describe how we used a genetic algorithms to tune the features that are used by the default inlining heuristic.

## 3.2 Program Characterization

Making an informed decision about inlining requires understanding the context from which a callee method is being called from. This information can be primarily be divided into two categories.

***Static Source Features*** Static source features are the features that are collected from the caller and the callee directly. This is done by categorizing each instruction in a method, and then counting the relative concentration of each of these instruction categories. These instruction categories are shown in Table 1.

***Information based on the call context*** This is the collection of information that is dependent on the particular call location, for example the current inlining depth, current graph size (this is the measure of the size of the code including the size of the past inlined calls.), whether the call is in a loop, or what is the relative probability of a particular block being executed. The last two factors are collected in the server compiler. The block weight feature can be calculated since the code being compiled already has some profiling information associated with it. In our case we make use of the

| 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 1.** The above are source feature categories collected from an inlining decision point. We collect features corresponding to the instructions for the caller and 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.

edge counts, each branch has counts of how many times the branch is taken, so for a conditional branch statement there are two targets from a single instruction, given how often each branch was taken we were able to calculate the probability of taking each branch. This information is also used as an input to the inlining oracle.

## 3.3 Applying NEAT

There are many characteristics (i.e., features) that can influence the inlining decision, and these factors may have complex interdependencies between them. In order to effectively model the nonlinear behaviour of these features, our neural networks are multi-layer perceptrons. In our neural networks, 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) [18] to construct an effective neural network to be used as an inlining heuristic. Figure 5 depicts the process of constructing a neural network using NEAT to replace the inlining heuristic in the Maxine VM.

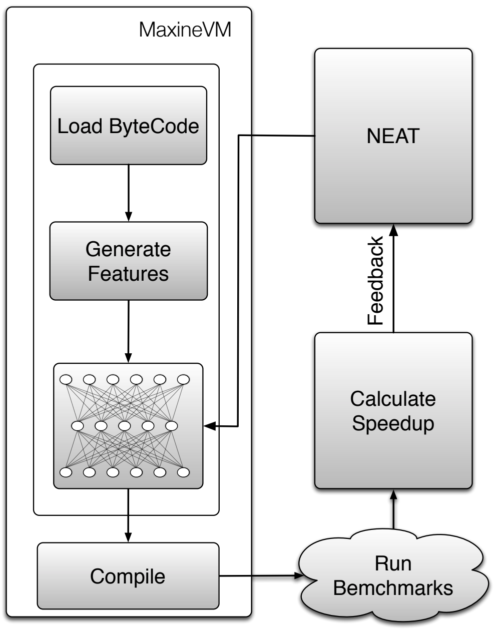

**Figure 5.** Framework used to construct effective inlining heuristics with neural networks using NEAT.

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 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 1 and produces an output indicating if inlining should be performed for a given call site.

## 3.4 Generating a Decision Tree

We show in Section 5 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. Alternatively, there are other machine learning techniques, such as decision trees, that can generate learned models that are easy to read and understand. However, the normal method of generating a decision tree cannot be applied directly in the case of building a decision tree for lack of a labelled dataset.

***Complications in generating dataset*** The C4.5 algorithm, a supervised learning algorithm, is used to generate the decision tree. It requires a fully labelled dataset. In supervised learning, the machine learning models is constructed based on a labelled training set. Each instance of this training set is a tuple consisting of an input set of features describing a decision point and a label corresponding to appropriate solution for that decision point. For the problem of inlining, a labelled data set is difficult to generate because individual inlining decision points are not independent of all other inlining decisions.

### Using NEAT networks to Construct Decision Tree Training Data

A labelled 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 labelled outputs. These labelled 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 [9] to generate a decision tree. This decision tree replaces the inlining heuristic in Maxine and is human-readable and easy to understand.

## 3.5 Tuning the Maxine VM with Genetic Algorithms

The Maxine VM contains an inlining heuristic that was manually constructed. Thus, the tests and features involved in this heuristic were devised from human intuition and empirical observations over a set of benchmarks running on a particular target architecture. There are four important features (described in Table 2) that make up this heuristic, and each of these features corresponds to a threshold value that can be fine-tuned to potentially improve this inlining heuristic. The possible space of threshold values for these features is large and exhaustive search is intractable. One method to fine-tune this heuristic is to use genetic algorithms to search over this space similar to previous work by Cavazos *et al.* [5].

Genetic algorithms provides a mechanism to intelligently traverse large search spaces. Genetic algorithms first construct an initial generation of randomly generated individuals. This population of individuals is evaluated using a fitness function and new individuals are progressively evolved over a series of generations to find the best individual based on a fitness function. For this paper, we use the ECJ toolkit [12], which provides several evolutionary algorithms.

| **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 Features | |
| `CALL_WARMTH` | A compund heuristic that is a combination of call invocation counts, estimated profit from inlining and estimated amount of work done in the callee |

**Table 2.** Features that are used by the default method inlining heuristic in the Maxine VM and the HotSpot VM.

For the problem of tuning an inlining heuristic, each individual consists of to a set of values to be used as thresholds in the heuristic as shown in Figure 3. For example, [10,30,5,100] correspond to ALWAYS_INLINE_SIZE = 10, CALLEE_MAX_SIZE = 30, MAX_INLINE_DEPTH = 5 and CALLER_MAX_SIZE = 100.

We use the values from each individual to run each benchmark in the training set, and we obtain an average performance for the entire training set. This average performance serves as the fitness for each individual. It is fairly straightforward to use GAs to fine-tune existing heuristics, however, using genetic algorithms to construct entirely new heuristic functions requires a fair amount of machine learning background to be effective.

### 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. Experimental Setup And Methodology

This section describes the benchmarks used for our training and test suites, the platforms experimented with, and the methodology we used for our experiments.

### 4.1 Benchmarks

We used a total of five benchmarks suites for training and testing: the Java Grande benchmarks [3], SPECjvm98 [16], SPECjvm2008 [17], SPECjbb2005 and DaCapo [2] and Scala Benchmarks [15] version 0.1.0.

***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 used *SPECjvm98* and *SPECjvm2008* benchmarks as our training set and used *Dacapo*, *SPECjbb2005* and *Scala* benchmarks for our test set.

### 4.2 Platform

***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 Xeopn X5680 CPUs running at 2.33 GHz with 8GBs of RAM.

### 4.3 Evaluation Methodology

We used two separate sets of benchmarks for our experiments. The training set was used to evolve and generate our machine learning-generated heuristics. The test set was used to evaluate our induced heuristics on a set of previously "unseen" programs. The training phase in the development of an inlining heuristic is expensive, but we envision this process to be performed "at the factory" where the JVM is being developed. Once the JVM has been tuned, it will be shipped to a customer where it will be used to compile and run code it has not "seen" before.

To evaluate an inlining heuristic on a benchmark, we measured the *run time* of each program. *Run time* refers to running time of the program without compilation time. The first iteration will cause the program to be loaded, compiled, and inlined according to the inlining heuristic being used. The remaining iterations will involve no compilation and will provide an accurate measure of a benchmark's run time.

| 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 3.** We used genetic algorithms to tune the existing inlining heuristic found in the Maxine VM. This table shows the default threshold values for the four inlining heuristic features and the tuned values found using genetic algorithms.

During the *training phase*, we ran each benchmark five times, and we used the minimum of the last three runs as the benchmark's run time. During the *testing phase*, we let each benchmark run for an initial warmup period of 20s to have all methods compiled. Then, we ran each benchmark four times, and we used the minimum of the last three runs.

## 5. Experimental 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.

### 5.1 Results on MaxineVM

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 the 250th generation.

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 6 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***

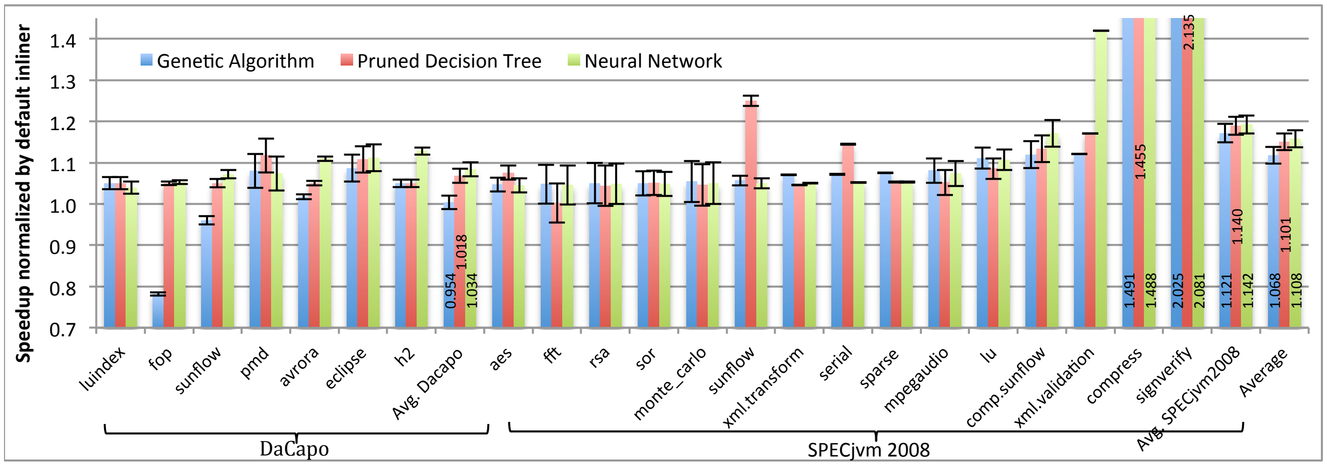

**Figure 6.** Performance of the three heuristics used in the paper. The heuristic tuned by the genetic algorithm is the present state of the art. In contrast, using the neural network heuristic constructed by NEAT as well as the pruned decision tree lead to better and more robust inlining heuristics. The results shown were based on using SpecJVM98, Java Grande and Jolden benchmarks as the training set and tested on the SPECjvm2008 and DaCapo benchmarks.

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 3.4. Once the training data was generated, we used the C4.5 decision tree algorithm in the Weka toolkit [9] 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 3 and 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 6 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 were no performance degradations for any of the benchmarks, except for fft which saw a degradation of 5%.

***Tuning Inlining Heuristics using Genetic Algorithms*** Cavazos *et al.* [5] 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 [12] to tune the Maxine inlining heuristic using GAs. We describe the process of tuning the inlining heuristic using GAs in Section 3.5. 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 3. Figure 6 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.

## 5.2 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.

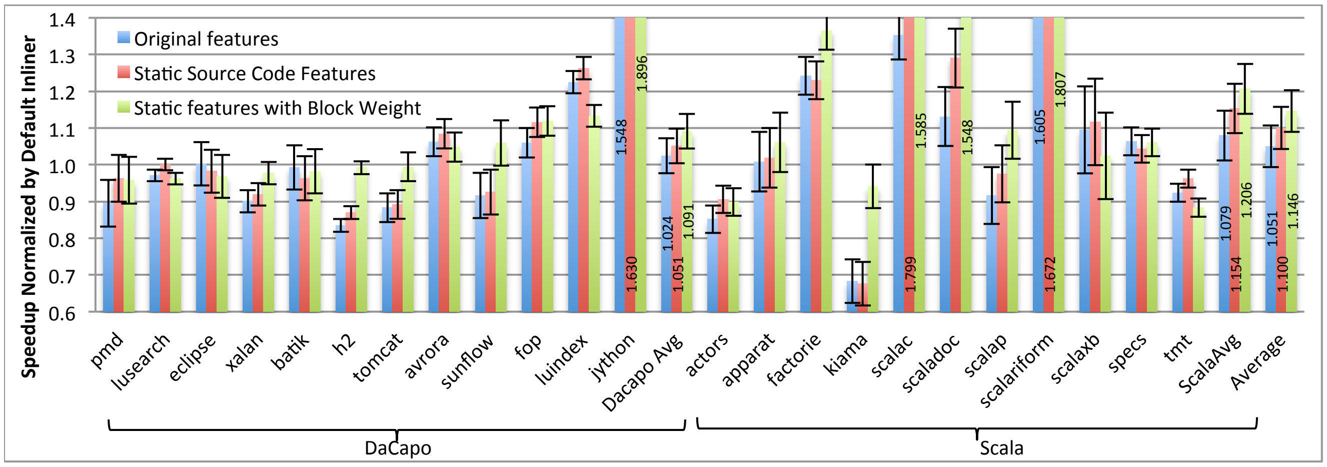

**Figure 7.** *Speedup of DaCapo and Scala Benchmarks*: performance of the network when provided with three different types of inputs. The first bar labelled "Original Features" includes features used in the default inlining heuristic, the second bar shows the performance when the neural network is provided with all source code features mentioned in Table 1 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.

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. In order to understand how good these features are we constructed three different NEAT heuristics using the three different feature sets.

***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 [15] 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 7.

***Original Features*** The first bar in Figure 7 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 1 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 7. 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 best results were achieved in this experiment where 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.

***Performance on SPECjbb2005*** The Java HotSpot VM is tuned specifically for the SPECjbb2005 benchmark. The results published on the website 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

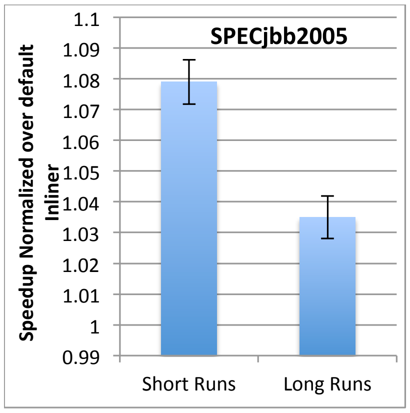

**Figure 8.** *Speedup of SPECjbb2005*: performance of the network when provided with all the static source code features along with Block Weight mentioned in Table 1. We used SPECjbb2005 as the training set and the test set.

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 SPECjbb2005 for the training set and show our results on the short runs and the long runs of SPECjbb2005. The short runs of the benchmark is run for 30 seconds for warmup 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 are shown in Figure 8, and we achieve speedups of around 8% and 3.5% for the short runs and the long runs, respectively.

### 5.3 Insights from Decision Tree

Generating a decision tree presents intriguing possibilities. The decision tree generated for the Java HotSpot VM using static features and block weight is shown in Figure 9. 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 seem to adjust the importance of the present inlining decision.

Given the height of the tree, we also see that the factors and the thresholds are dependent on other factors 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.

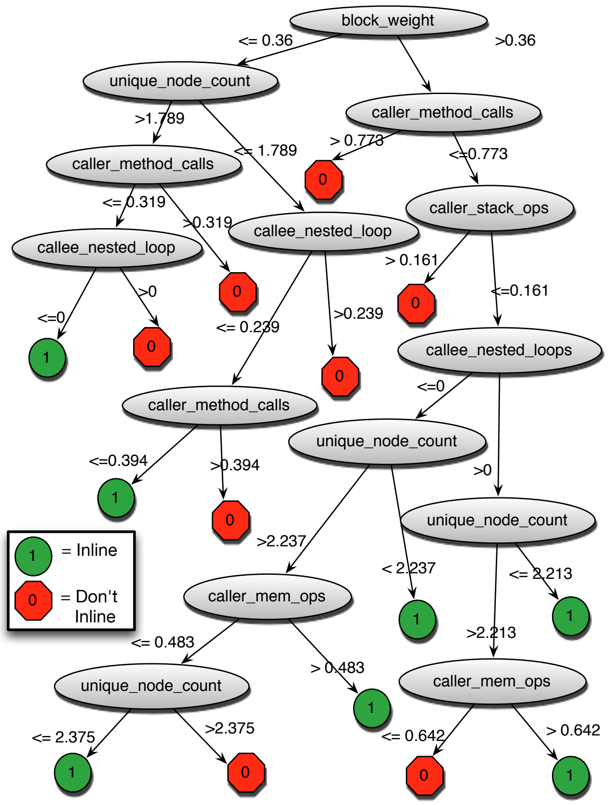

**Figure 9.** The pruned decision tree that provides an insight on what features were useful in making inlining decisions

## 6. Related Work

In this section we review the most relevant work related to the inlining and machine learning.

***Method Inlining*** Cooper *et al.* [6] 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.

Arnold *et al.* [1] represents the inlining problem as a knapsack problem that calculates the size/speed tradeoffs 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.* [10] 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.* [8] 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.

Leupers *et al.* [11] 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.

***Machine Learning*** Stephenson *et al.* [19] used genetic programming (GP) to tune heuristic priority functions for three compiler optimizations: hyperblock selection, register allocation, and data prefetching within the Trimaran's IMPACT compiler. For two optimizations, hyperblock selection and data prefetching, they achieved significant improvements. Genetic programming is a branch of genetic algorithms that evolves a population of parse trees. For this research, each parse tree was an expression that represents a priority function. To start, Stephenson *et al.* generated random expressions that represented priority functions for a particular compiler optimization. Each expression was given a fitness based on its performance and then crossovers and mutations were performed by modifying the expressions with relational and/or real-valued functions.

Cooper *et al.* [7] use genetic algorithms to solve the compilation phase ordering problem. They were concerned with finding "good" compiler optimization sequences that reduced code size. Unfortunately, their technique is application-specific. That is, a genetic algorithm has to *retrain* for each program to decide the best optimization sequence for that program. Their technique was successful at reducing code size by as much as 40%.

Cavazos *et al.* [4] describe an idea of using supervised learning to control whether or not to apply instruction scheduling. They induced heuristics that used features of a basic block to predict whether scheduling would benefit that block or not. Using the induced heuristic, they were able to reduce scheduling effort by as much as 75% while still retaining about 92% effectiveness of scheduling all blocks.

Monsifrot *et al.* [13] worked on trying to predict the benefits of unrolling using a classification decision tree. They worked with Fortran programs on two different architectures, and noticed an improvement of around 3%.

## 7. Conclusions

Inlining is an important optimization for many programming languages. However, it is not always apparent what the most important features are for this optimization or the relative importance of these features. We describe a novel technique of characterizing the code being compiled and using these characteristics to construct inlining heuristics automatically. We used neural networks constructed using neuro-evolutionary algorithms and decision trees to construct excellent inlining heuristics that were able to obtain significant improvement over hand-tuned heuristics. On our test sets, we achieved 11% on average on the Maxine VM and 15% on average on the Java HotSpot VM. We also showed how to construct concise and readable heuristics with decision tree algorithms by using our neural networks to construct the training set for these supervised learning algorithms. Our results show that it is beneficial to use a variety of different caller, callee, and context sensitive features to construct inlining heuristics using machine learning.

## References

[1] 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.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[3] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. *Concurrency - Practice and Experience*, 12(6):375–388, 2000.

[4] J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 183–194, Washington, D.C., June 2004. ACM Press.

[5] J. Cavazos and M. F. P. O'Boyle. Automatic tuning of inlining heuristics. In *IN ACM/IEEE CONFERENCE ON SUPERCOMPUTING*, page 14. IEEE Computer Society, 2005.

[6] K. Cooper, T. Harvey, and T. Waterman. An adaptive strategy for inline substitution. In L. Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78791-4_5.

[7] K. D. Cooper, P. J. 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, Atlanta, Georgia, July 1999. ACM, ACM Press.

[8] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *LISP and Functional Programming*, pages 273–282, 1994.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[10] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *First Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 253–264, San Francisco, CA, March 2003.

[11] 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*, pages 253–256, Piscataway, NJ, USA, 1999. IEEE Press.

[12] S. Luke. ECJ 11: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2004.

[13] A. Monsifrot and F. Bodin. A machine learning approach to automatic production of compiler heuristics. In *Tenth International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA)*, pages 41–50, Varna, Bulgaria, September 2002. Springer Verlag.

[14] M. Paleczny, C. Vick, and C. Click. The Java HotSpot$^{TM}$ server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*, pages 1–12. USENIX, 2001.

[15] 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 *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM.

[16] Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. *SPEC JVM98 Benchmarks*, 1998.

[17] Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. *SPEC JVM2008 Benchmarks*, 2008.

[18] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[19] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 77–90, San Diego, Ca, June 2003. ACM Press.

[20] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynes, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 2013.