# Automatic Tuning of Inlining Heuristics

John Cavazos Michael F.P. O'Boyle
Member of HiPEAC
Institute for Computing Systems Architecture (ICSA), School of Informatics
University of Edinburgh, United Kingdom

## Abstract

*Inlining improves the performance of programs by reducing the overhead of method invocation and increasing the opportunities for compiler optimization. Incorrect inlining decisions, however, can degrade both the running and compilation time of a program. This is especially important for a dynamically compiled language such as Java. Therefore, the heuristics that control inlining must be carefully tuned to achieve a good balance between these two costs to reduce overall total execution time. This paper develops a genetic algorithms based approach to automatically tune a dynamic compiler's internal inlining heuristic. We evaluate our technique within the Jikes RVM [1] compiler and show a 17% average reduction in total execution time on the SPECjvm98 benchmark suite on a Pentium-4. When applied to the DaCapo benchmark suite, our approach reduces total execution time by 37%, outperforming all existing techniques.*

## 1 Introduction

Java™is a highly portable programming language and is the language of choice in adaptive distributed grid environments. However, this portability is often at the cost of poor performance [8] and a deterrent to wider usage in high performance computing. This has encouraged compiler researchers to develop optimizations that improve performance without sacrificing its portability.

One of the major areas of interest has been in using method inlining as a performance enhancing optimization. It can significantly reduce execution time because it reduces calling overhead and has the potential to increase opportunities for further optimizations.

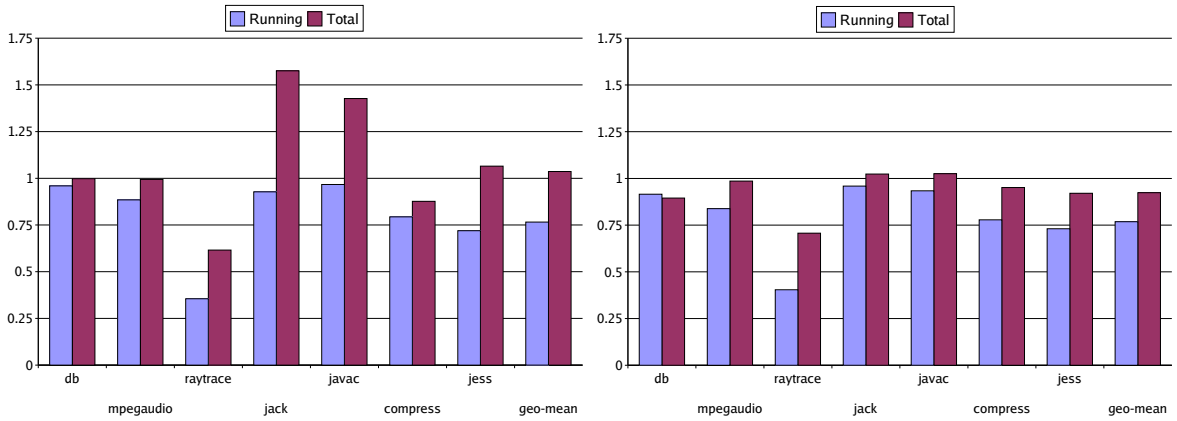
Overly aggressive inlining, however, can have a negative impact on a program's performance due to a larger runtime memory footprint and increased I-cache misses. It can also lead to increased compilation time and memory overhead. This is especially important in the context of dynamically compiled programming languages, such as Java and C# where compilation occurs at runtime and is part of the total execution time of a program. Thus, inlining heuristics must be carefully tuned to increase inlining benefits while avoiding its potential costs.

Determining whether or not to inline a method is a difficult compiler decision and depends on the calling context of the method, the processor it is running on and the other optimizations which the compiler considers. This decision increases in difficulty in the case of Just-In-Time (JIT) compilation as the potential benefits may be outweighed by increased compilation time. So, as with many compiler optimizations, a heuristic typically controls when and to what extent inlining should be applied when compiling a program. These heuristics are carefully tuned by compiler experts; a difficult and time-consuming task. However, the heuristics are inherently static, hard-coded into the compiler and do not take account changing environment or platform. In the case of Jikes™Research Virtual Machine (Jikes RVM), the compiler considered within this paper, for instance, the same inline heuristic is used for both the Intel and PowerPC architectures.

This paper develops a machine learning based technique that automatically derives a good inlining heuristic. In effect, it automates part of the compiler expert's role by tuning a heuristic across a large training set automatically. This job is performed just once, off-line, each time the compiler is ported to a new platform and is then incorporated into the compiler to be used on future application codes.

Optimizing inlining in the case of dynamic compilation adds another degree of complexity, that of compilation scenario. Dynamic compilers frequently provide the programmer with a range of optimization scenarios: from fast non-optimizing compilation right up to longer full aggressive optimization. The reason is that as compilation time is part of execution time, there are occasions where it may be important to reduce the cost of compilation rather than running time to reduce the overall cost of a program run or vice-versa.

In addition dynamic compilers support adaptive or "hot




(a) Effectiveness of Inlining for Optimizing Compiler.

(b) Effectiveness of Inlining for Adaptive Compiler.

**Figure 1. Relative time reduction with inlining. Total time = Running time + dynamic compilation time**

spot" based compilation where the program is initially compiled with a fast non-optimizing compiler and later, when a frequently used method or hot spot is detected, an optimizing compiler is used to recompile it. This attempts to provided a means of balancing compilation and running time. In fact we show, in the next section, that the optimization scenario that achieves the shortest execution time varies from program to program. So, as well as the issue of adapting optimization heuristics to platform architecture, we would like the heuristic to be specialized to the particular compilation scenario chosen by the programmer on the command-line. Again, in practice, a single heuristic is currently used across different optimization scenarios.

Clearly the "one-size-fits-all" heuristic is not the best for each of these compilation scenarios. Instead this paper develops a technique that automatically determines the best inlining heuristic for each compilation scenario. This is achieved by using an off-line machine learning algorithm based on a genetic algorithm. When incorporated into the Jikes RVM compiler, we show that our approach outperforms the best previously known heuristic by 37% on the SPECjvm98 benchmarks. Thus our approach develops inlining heuristics for new platforms, with no human intervention, that are better than hand-crafted approaches. It specializes the heuristic to the particular compilation scenario chosen by the programmer and is therefore able to outperform existing approaches across all platforms, benchmarks and compilation options.

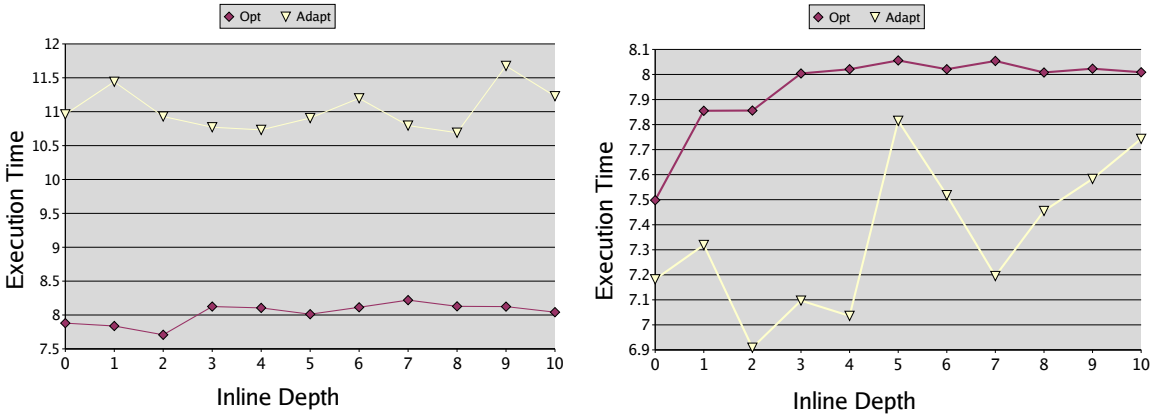
The paper is structured as follows, section 2 provides a motivating example showing the impact of inlining and how it is difficult to determine the best compiler heuristic. Section 3 outlines the various optimization scenarios considered in this paper and describes our machine learning based approach to determine the heuristic automatically. Section 4 describes the experimental setup and is followed in section 5 by the methodology used. Section 6 analyses the results and is followed in section 7 by a review of related work. This is followed by some concluding remarks in section 8.

## 2 Motivation

This section shows how inlining heuristics have a varying impact on performance and are sensitive to the program, platform and compilation scenario.

**Compilation scenario** Inlining is one of the most important optimizations for several programming languages. It can have a substantial impact, but incorrectly inlining a method can have a detrimental effect. Figures 1(a) and 1(b) show the impact of enabling the existing inling heuristic in the Jikes RVM compiler across the standard SPECjvm98 suite of Java benchmarks for two different compilation scenarios [1], full optimization (*Opt*) and adaptive optimization (*Adapt*). *Opt* always uses an aggressive optimization level when compiling and is therefore likely to take longer to compile but produces faster code. Adaptive compilation initially uses a fast non-optimizing compiler and dynamically recompiles with higher optimization levels if the method is a hot-spot. As compilation time is dynamic and part of the overall execution time, it attempts to strike a balance be-

[1] We will discuss the different compilation scenarios in more detail in Section 3.3




(a) Varying Inlining Depth for Compress.

(b) Varying Inlining Depth for Jess.

**Figure 2. Execution time vs Inlining Depth for 2 SPECjvm98 benchmarks.**

tween compilation and running time to minimize overall or total execution time.

We therefore give two performance results: *running* time and *total* time. Running time indicates the benchmark running times *without* compilation time while total time indicate running times *with* compilation. The values are normalised to running and total times without inlining so bars below 1 indicate performance improvement while bars over 1 indicate performance degradation.

For the full optimization scenario in Figure 1(a), Jikes RVM is able to achieve a substantial improvement in *running* time for several benchmarks giving an average improvement of 24%. However, this comes at the expense of increased compilation time, which causes a large degradation for two programs, leading to an an average degradation of 3% in total execution time. Clearly the compiler developers have focussed on reducing running time but here the inliner is overly aggressive and actually slows down programs on average due to the excessive compilation time. A more balanced approach is needed.

Under an adaptive compilation scenario the Jikes RVM inlining heuristic is almost always beneficial (it degrades the total time of two programs) with an average improvement in of 23% for running time and 8% for total time. These graphs show the importance of inlining and that one heuristics is not ideal for differing scenarios.

**Parameter Sensitivity** It is, in fact, very difficult to determine the right inlining heuristic. The performance of inlining is highly sensitive to the heuristic controlling it. Consider Figure 2(a) and 2(b) which shows the execution time in seconds of two programs when inlining is enabled and we vary just one of the parameters that controls the inlining decisions in Jikes RVM: inline depth. This is performed for the two different compilation scenarios, *Opt* and *Adapt*.

First of all, it is worth noting that the choice of compilation scenario that is best to use is not always obvious. For the first program compress, *Opt* is best and for the second, jess *Adapt* is the best choice. This explains the need for supporting multiple compilation scenarios in dynamic compilers.

Inline depth controls the maximum depth of the call chain that will be considered for inlining at a particular callsite and we vary the value of this parameter from 0 to 10 for the two different programs. The default value used in Jikes RVM for this parameter is 5. These two graphs indicate that 5 is not the right value to use for both these benchmarks under both compilation scenarios. For compress, 2 is a better value to use under *Opt* and 8 is better for *Adapt*. For jess, 0 is the best value for *Opt* and 2 is the best value for *Adapt*. Interestingly, the Jikes RVM value of 5 is the worst option for both compilation scenarios for jess.

As can be seen by these graphs inlining is an important optimization that can result in substantial increases in benchmark performance. However, selecting the right values that control inlining are important to achieving the best performance from this optimization.

In Jikes RVM, there is generally a fixed heuristic for inlining regardless of the compilation scenario Also, there is no mechanism of telling the compiler that it should optimize

toward a different goal, such as reducing running time or the total execution time. In Section 6, we show results for machine learning derived heuristics for different compilation scenarios, for different platforms targetted at different goals i.e. reducing running time vs. total time. In each case our automatically derived optimization outperforms the manually tuned heuristic.

## 3 Problem and Approach

Manually fine tuning compiler heuristics is a tedious and complex task. It can take a compiler writer many weeks to fine tune a heuristic and once a new architecture is targetted or an optimization is added to the compilation system, the tuning process must be repeated again. Machine learning automates this search process and is a desirable alternative to manual experimentation. Our approach uses a genetic algorithm to fine tune the inlining heuristic.

The first step in applying genetic algorithms to this problem requires discovering what parameters control the particular optimization of interest. Typically, fine tuning optimization heuristics requires experimenting with several different parameters and parameter values. Jikes RVM, for instance, has four or five parameters depending on the compilation scenario being used. Table 1 describes each parameter and the range of values we experimented with. Given the large range of values we were looking at ($3 \times 10^{11}$), exhaustive search is intractable. Genetic algorithms provides a mechanism that can intelligently search this large space of values.

The second step is to phrase the problem of tuning a heuristic as a genetic search problem. This entails being able to experiment with several parameter values used by the heuristic and measuring the performance of the particular setting of these values in order to provide feedback for the genetic algorithm.

We emphasize that these steps of tuning the heuristic happen *off-line*. We are simply replacing the manual ad hoc tuning process of a compiler heuristic with an automatic approach that searches a large space of parameter values. Once a heuristic is tuned, the compiler is delivered with a fixed set of values for each different compilation scenario or architecture of interest and there is no further evolving of these parameter values. Thus there is no runtime overhead associated with this approach.

### 3.1 Applying Genetic Algorithms

We needed an machine learning algorithm that could search the large space of parameter values efficiently and that could use a program's performance to guide the search. Genetic algorithms are a good candidate. Here we describe genetic algorithms and then the specifics to how we applied them to this problem.

Genetic algorithms start with a randomly generated number of individuals. This population of individuals is then progressively evolved over a series of generations to find the best individual based on a fitness function. The evolutionary search uses the Darwinian principle of natural selection (survival of the fittest) and analogs of various naturally occurring operations, including crossover (sexual recombination), mutation, gene duplication, gene deletion.

The genetic algorithm used is called ECJ [11]. This system is written entirely in Java and has a wide variety of algorithms and options with which it can be configured. We start with a set of randomly generated individuals, where each individual is a vector of integers representing the different values of the parameters controlling the inlining heuristic. For example, one individual could be $[10, 15, 5, 2000, 300]$ representing CALLEE_MAX_SIZE=10, ALWAYS_INLINE_SIZE=15, MAX_INLINE_DEPTH=5, CALLER_MAX_SIZE=2000, and HOT_CALLEE_MAX_SIZE =300. ECJ allows the specification of a range of values (cf. Table 1) to search over for each parameter. An initial population size of 20 was selected and this population was evolved over 500 generations. Each individual (or set of parameter values) is given a fitness value which indicates how "good" a particular set of parameter values is.

#### Fitness Functions

The fitness value used was the geometric mean of the performance of the SPECjvm98 benchmarks. That is, the fitness value for a particular performance metric is:

$$Perf(S) = \sqrt[|S|]{\prod_{s \in S} Perf(s)}$$

where $S$ was the benchmark training suite and Perf($s$) was the metric to minimize for a particular benchmark $s$.

The metrics we are interested in minimizing are: running time, total time, or a *balance* of both. When optimizing for *balance* the following formula for Perf($s$) was used.

$$Perf(s) = factor * Running(s) + Total(s)$$

where $factor = Total(s_{def})/Running(s_{def})$ and $s_{def}$ is a run of benchmark $s$ using the default heuristic.

### 3.2 Inlining Heuristic in Jikes RVM

Figure 3 shows a high-level view of the inlining heuristic used in Jikes RVM. This heuristic decides whether or not to inline based on a series of tests. The first two tests pertain to the *estimated* size of the method being considered

| Inlining Parameters | Description | Range |
|---|---|---|
| CALLEE_MAX_SIZE | Maximum callee size allowable to inline | 1−50 |
| ALWAYS_INLINE_SIZE | Callee methods less than this size are always inlined | 1−25 |
| MAX_INLINE_DEPTH | Maximum inlining depth at a particular call site | 1−15 |
| CALLER_MAX_SIZE | Maximum caller size to inline into | 1−4000 |
| HOT_CALLEE_MAX_SIZE | Maximum hot callee to inline | 1−400 |

**Table 1. Parameters That Were Tuned With Genetic Algorithms.**

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

**Figure 3. Optimizing Inlining Heuristic**

for inlining (callee). This size is an estimate of the number of machine instructions that will be generated for the method. The first test restricts methods that are particularly large (greater than the parameter CALLEE_MAX_SIZE) from being inlined. The second test causes small methods (methods less than the parameter ALWAYS_INLINE_SIZE) to be inlined. These small methods should generate less code than the calling sequence and parameter setup if the call was left in place. The third test imposes an limit on the maximum depth of the inlining decisions at any given call site. The forth test checks whether the estimated size of the caller method is large. If the caller method is larger than a particular value (CALLER_MAX_SIZE), we restrict inlining and further code expansion to avoid excessive compilation times. Finally, if all tests are false, the callee method is inlined.

Under an adaptive scenario, profiling is used to discover calls that are frequently executed. When a method is recompiled, a call site is checked whether it is hot or not. If the call site is *not* hot, the inlining tests described in the previous paragraph are used. If the call site is hot, a single test (described in Figure 4) is used to check whether the estimated callee size exceeds a particular value (HOT_CALLEE_MAX_SIZE). If it does not exceed this threshold, then the method is inlined.

### 3.3 Compilation Scenarios

When running Java programs there are several compilation scenarios that one can choose from. We ran experiments under two popular Java compilation scenarios: Adaptive and Optimizing.

```
inlineHotCallSite(calleeSize)
    if (calleeSize > HOT_CALLEE_MAX_SIZE)
        return NO;
    return YES;
```

**Figure 4. Adaptive Inlining Heuristic**

Under the Adaptive (*Adapt*) scenario, all dynamically loaded methods are first compiled by the non-optimizing baseline compiler that converts bytecodes straight to machine code without performing any optimizations, not even inlining. 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 and calls in the methods are subject to the inlining heuristic [2]. Under the Optimizing (*Opt*) scenario, an optimizing compiler is used to compile all methods. This scenario is typically used for long running programs where compilation is a small fraction of the program's running time. This scenario is also of interest when running short running programs where the program will not run long enough for profiling to return useful information.

**Total vs Balance** As well as the two different compiler scenarios *Adapt* and *Opt* we also considered different optimization goals namely: total time, or a *balance* between running and compilation time. As compilation is part of the total execution time for dynamic compilers then optimizing for total time will try to minimize their combined cost. It may be the case however, that reducing total cost by, say, reducing compilation time may be at the expense of increasing running time, which is not always advantageous. Instead, when the program is likely to run for a considerable length of time, it may be preferable for the user to reduce the running time at the expense of potentially greater compilation time. In between these two cases, we have probably the most useful case of *balance*, where we aim to reduce the

| Program | Description |
|---|---|
| compress | Java version of 129.compress from SPEC 95 |
| jess | Java expert system shell |
| db | Builds and operates on an in-memory database |
| javac | Java source to bytecode compiler in JDK 1.0.2 |
| mpegaudio | Decodes an MPEG-3 audio file |
| raytrace | A raytracer working on a scene with a dinosaur. This is the single-threaded variant of mtrt |
| jack | A Java parser generator with lexical analysis |

**Table 2. Characteristics of the SPECjvm98 benchmarks.**

| Program | Description |
|---|---|
| antlr | parses one or more grammar files and generates a parser and lexical analyzer for each |
| fop | takes an XSL-FO file, parses it and formats it, generating a PDF file |
| jython | inteprets a series of Python programs |
| pmd | analyzes a set of Java classes for a range of source code problems |
| ps | reads and interprets a PostScript file |
| ipsixql | Performs a query against the complete works of William Shakespeare. |
| pseudojbb | SPECjbb2000 modified to perform fixed amount of work. Executes 70000 transactions for one warehouse. |

**Table 3. Characteristics of the DaCapo+JBB benchmarks.**

total execution time without excessively increasing running time.

# 4 Experimental Setup

## 4.1 Benchmarks

We examine two suites of benchmarks. The first is the SPECjvm98 suite [14] detailed in Table 2. These were run with the largest data set size (called 100). We used the single-threaded version of mtrt, called raytrace, to get easily reproducible results.

The second set of programs consists of 5 programs from the DaCapo benchmark suite [13] version beta050224 and two additional benchmarks: ipsixql and SPECjbb2000 all described in Table 3. The DaCapo benchmark suite is a collection of programs that have been used for various different Java performance studies aggregated into one benchmark suite. We ran the DaCapo benchmarks under its default setting. We also included ipsixql a real-world application that implements an XML database and a modified version of SPECjbb2000 (hence, we call it pseudojbb) that performs a fixed number of transactions (instead of running for a predetermined amount of time) to more easily observe effects of inlining.

## 4.2 Platforms

We tuned our inlining heuristics in the Jikes Research Virtual Machine [1] version 2.3.3 for two different architectures: an Intel and a PowerPC architecture. The Intel processor is a 2.8 GHz Pentium-4 based Red Hat Linux workstation with 500M RAM and a 512KB L1 cache. The PowerPC architecture is an Apple Macintosh system with two 533 MHz G4 processors, model 7410 with 640M RAM and 64KB L1 cache. Both these processors are aggressive superscalar architectures and represent the current state of the art in processor implementations. We used the OptAdaptiveSemispace configuration of Jikes RVM, indicating that the core virtual machine was compiled by the optimizing compiler, that an adaptive optimization system in included in the virtual machine, and that the basic semispace copying collector was used.

# 5 Evaluation Methodology

As is customary our learning methodology was to tune over one suite of benchmarks, commonly referred to in the machine learning literature as the *training* suite. We then test the performance of our tuned heuristic over another ”unseen” suite of benchmarks, that we have not tuned for, referred to as the *test* suite. This makes sense in our case for following reason. We envision developing and installing of the heuristic “at the factory”, and it will then be applied to code it has not “seen” before. To evaluate an inlining heuristic on a benchmark, we consider two kinds of results: *total time* and *running time*.

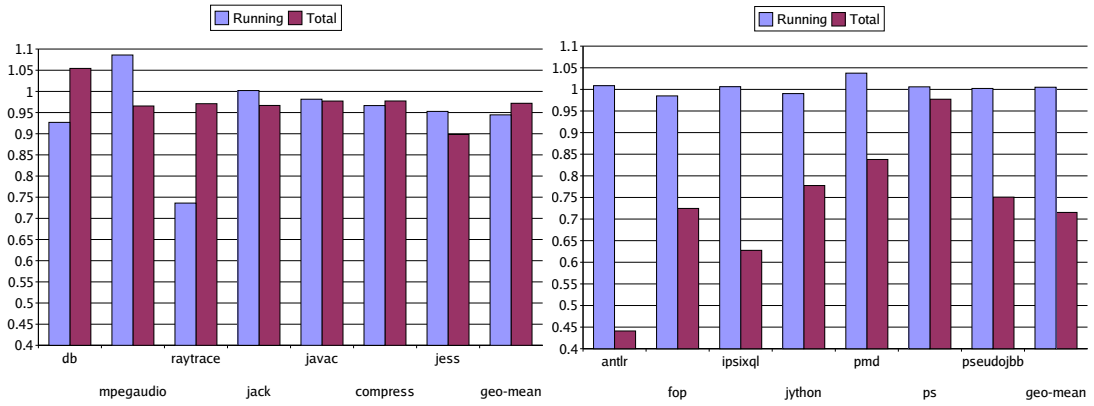
*Total time* refers to the running time of the program including compilation time.

*Running time* refers to running time of the program without compilation time.

To obtain these numbers, we requested that the Java benchmark iterate at least twice. The first iteration will cause the program to be loaded, compiled, and inlined according to the appropriate inlining heuristic. We used this iteration as our total time measure. The remaining iterations should involve no compilation; we use the best of the remaining runs as our measure of running time.

| Parameters | Compilation Scenarios | | | | | |
|---|---|---|---|---|---|---|
| | Default | Adapt | Opt:Bal | Opt:Tot | Adapt (PPC) | Opt:Bal (PPC) |
| CALLEE_MAX_SIZE | 23 | 49 | 10 | 10 | 47 | 23 |
| ALWAYS_INLINE_SIZE | 11 | 15 | 16 | 6 | 10 | 11 |
| MAX_INLINE_DEPTH | 5 | 10 | 8 | 8 | 2 | 8 |
| CALLER_MAX_SIZE | 2048 | 60 | 402 | 2419 | 1215 | 240 |
| HOT_CALLEE_MAX_SIZE | 135 | 138 | NA | NA | 352 | NA |

**Table 4. Inlining Parameter Values Found for Intel x86 and PowerPC (PPC)**




(a) Performance on SPECjvm98

(b) Performance on DaCapo+JBB

**Figure 5. Reduction in time relative to Jikes RVM heuristic. Adaptive scenario tuned for balance on x86**

## 6 Experimental Results

We now consider the quality of the inline heuristic produced by our genetic algorithms. We run experiments under different compilation scenarios and two different architectures. Initially we consider the adaptive optimization scenario (*Adapt*) on the x86 platform. As this scenario is aimed at balancing the cost of compilation with that of running time, with careful hot spot based recompilation, we only consider optimizing for balance here. Next, we consider the optimizing *Opt* scenario, again for the x86, for both balanced optimization (*Opt:Bal*) and reducing total execution time (*Opt:Tot*). The experiments are repeated on the PowerPC for balanced optimization under *Adapt* (*Adapt(PPC)*) and *Opt* (*Opt(PPC)*) compilation scenarios. Finally, to show the flexibility of our approach we consider tuning the heuristic on each program in order to reduce *running* time on the x86.

For each scenario, we use genetic algorithms to tune the inlining heuristic for the SPECjvm98 benchmarks. We then use the tuned heuristic to compile our test benchmarks, DaCapo+JBB. We present the results for both benchmarks suites. For each set of results, we compare the running and total time reduction using our automatically tuned heuristic versus the manually tuned heuristic found in Jikes RVM. Bars below 1 indicate a performance improvement over the default heuristic while bars over 1 indicate a performance degradation.

### 6.1 Tuned Parameter Values

Table 4 shows the parameter values found by the genetic algorithm for each compilation scenario and for each architecture. The first column also shows the default values shipped with Jikes RVM. Notice that values found by our technique vary widely among the different compilation scenarios and architectures. For both architectures, *Adapt* tends to have larger values for the parameter CALLEE_MAX_SIZE compared to *Opt*. However, for the Intel architecture it is preferable to use smaller values under *Adapt* for parameter CALLER_MAX_SIZE compared to *Opt*, while on the PowerPC the inverse is true. The table also shows that for both architectures the default values for CALLER_MAX_SIZE are overly aggressive. This will have a tendancy to cause unecessarily large compile times which is consistent with our results.

The table shows a wide range of values for the parameter
MAX_INLINE_DEPTH. This parameter restricts the

amount of inlining at any one call site, thus restricting the limit of growth to each method. For Adapt, larger values are preferable on the Intel architecture while smaller values are preferable for PowerPC. This may be due to the smaller L1 cache size of the PowerPC which may bias towards smaller footprints. The values found for the parameter ALWAYS_INLINE_SIZE are pretty consistent among compilation scenarios and architectures (ranging from 9 to 16). This is to be expected as methods that are always beneficial to inline should be relatively small compared to the call and parameter setup. We now discuss the results for each compilation scenario and architecture.

## 6.2 Adaptive Balanced x86 Scenario

The first compilation scenario we experiment with is the Adaptive *Adapt* balanced optimization scenario on the x86. Figure 5(a) shows the performance of our tuned heuristic on SPECjvm98. Our tuned heuristic improves the running time of 5 benchmarks, obtaining a significant reduction of 27% for raytrace. However, on mpegaudio we degrade performance by 8%. On average, we obtain a good running time reduction of 6% over the default heuristic. Our tuned heuristic also obtains reductions in total time (with compilation) for several benchmarks, as much as a 10% reduction on jess. Overall we get an average reduction in total time by 3% showing that inlining is a well-studied optimization in Jikes RVM and the default heuristic has been well-tuned especially for the SPECjvm98 benchmark suite.

Figure 5(b) shows no significant reductions or degradations in running time for the DaCapo+JBB test suite and, on average, we achieve the same running time as the default heuristic. However, we obtain impressive reductions in total time on 6 out of 7 of the benchmarks by up to 56%. On average, we obtain an 29% reduction in total time for these benchmarks.

Our heuristic performs well on SPECjvm98 benchmarks which is to be expected as it was tuned over it. However, it is suprising that we achieved such a significant reductions in total time for our test benchmarks as our heuristic was not tuned for these. Furthermore, we can reduce total time in a compilation scenario, *Adapt* whose goal is already to reduce total time. Clearly, the correct tuning of an inlining heuristic can have a large impact on performance.

## 6.3 Optimizing Compilation Scenario

The second compilation scenario we investigate is *Opt*. As previously mentioned, under this scenario all methods that are dynamically invoked are compiled with an aggressive optimizing compiler. We performed two different tuning experiments. Tuning for a good balance and tuning for reduction in total time.

**Tuning for Balance**

Figure 6 (a) shows our results for tuning the inlining heuristic to achieve a good balance of running time and total time for *Opt* in SPECjvm98. These results show we can improve running time for several benchmarks under this scenario. On average, we obtain a 4% reduction over the default heuristic. Because we are tuning for a balance, the genetic algorithm may allow some degradation in running time if it can achieve reductions in total time. This leads to an significant average reduction in total time of 16%.

For the test suite, DaCapo+JBB, in Figure 6 (b), we see even better performance using our tuned heuristic. Our heuristic obtains large reductions in running time for a 2 benchmarks. We do get degradations on 4 benchmarks, however, on average we obtain a modest improvement in running time of 3%. We obtain larger reductions in total time over the default heuristic and on average, we obtain almost a 26% reduction in total time over the default heuristic on these benchmarks.

Our tuned heuristic performs well on the training suite as well as on our test suite of benchmarks that we had not tuned for. We speculate that the default heuristic has been tuned especially for SPECjvm98, given their popularity as a benchmark suite for testing the performance of Java compilers.

**Tuned For Total Time**

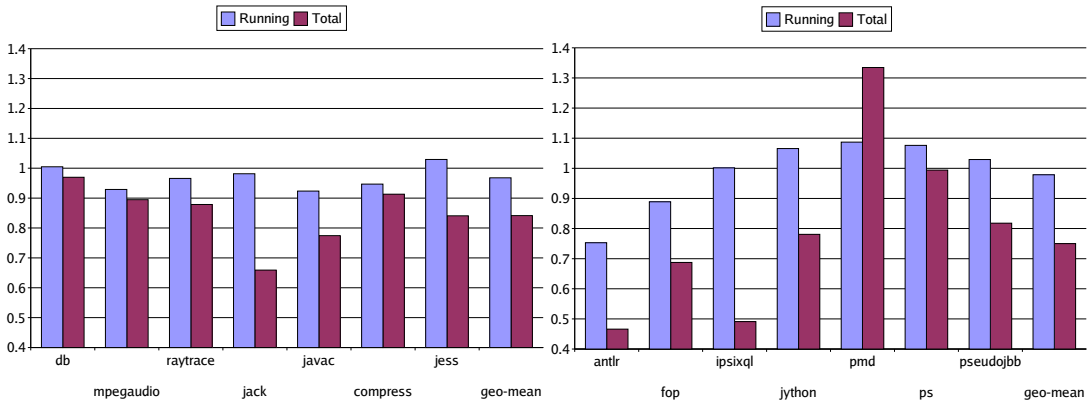
Under the third scenario, we tune our inlining heuristic to obtain the best total times, shown in Figure 7 [2]. Intuitively, the tuned heuristic should be a less aggressive one, not degrading the running time of programs too much, but decreasing compile time (and therefore total time) substantially. Here, we might be able to achieve a further improvement in compiling time over the improvement we achieved when tuning for a balance. On average, we can achieve a reduction in total time of 17% with a slight reduction in running time on average of 1%.

The result for DaCapo+JBB benchmark suite are even more favorable. We reduce average total time by more than 35% with a small (4%) increase in running time. This small increase in running time is to be expected since we are optimizing for total time. On several benchmarks in this suite, we can achieve dramatic reductions in total time (35% for fop, 46% for pseudojbb, 50% for ipsixql, and 58% for antlr).

## 6.4 Tuned for a Different Architecture

We repeated two of our tuning experiments for the PowerPC archecture to see how the values of our inlining parameters would change and what improvements we could

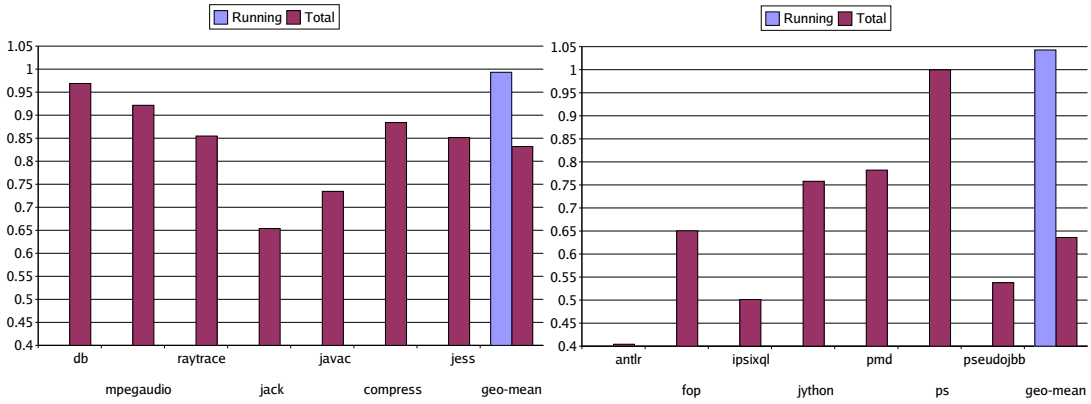
[2] As we are optimizing for total time, only the average running time is given




(a) Performance on SPECjvm98

(b) Performance on DaCapo+JBB

**Figure 6. Optimizing scenario, *Opt*, tuned for balance on x86 (Opt:Bal)**




(a) Performance on SPECjvm98

(b) Performance on DaCapo+JBB

**Figure 7. Optimizing scenario, *Opt*, tuned for Total execution time on x86 (Opt:Tot)**

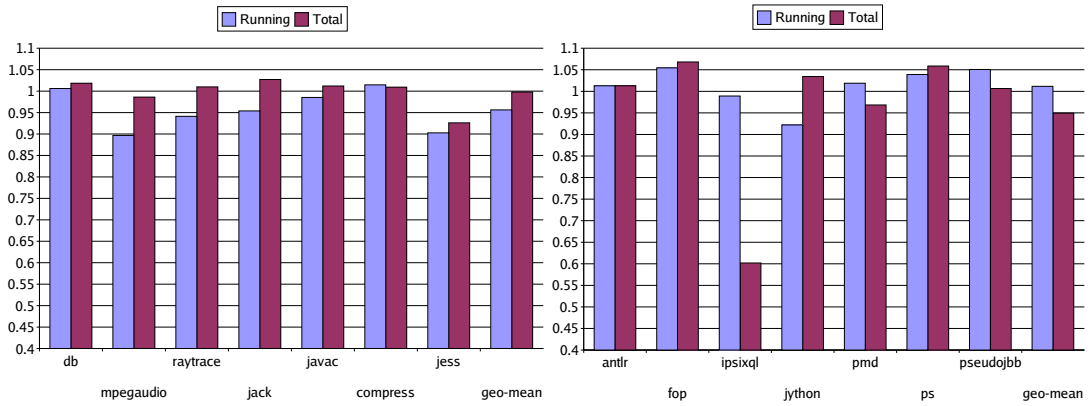
achieve over the default heuristic. We tuned the heuristic for the Adaptive and the Optimizing compilation scenarios, both tuned for balance.

### Adaptive Compilation Scenario

In Figure 8, we present results from tuning our heuristic for *Adapt* for the PowerPC. Our results show we can reduce running time for several programs, by at least 10% on mpegaudio and jess and by 5% or more for jack and raytrace. On average we get a good reduction in running time of 5%. For total time, the only benchmark we significantly reduce is jess by 8%. For the other benchmarks we either have a slight increase or decrease or no change at all. On average we achieve a 1% reduction in total time. Again, since we are tuning for balance this is to be expected as the genetic algorithm finds a heuristic where there is the most to be gained on average from running time and total time. In the case of the DaCapo+JBB suite, we have the inverse result where we suffer a 1% increase in running time but reduce total execution time by 6%.

### Optimizing Compilation Scenario

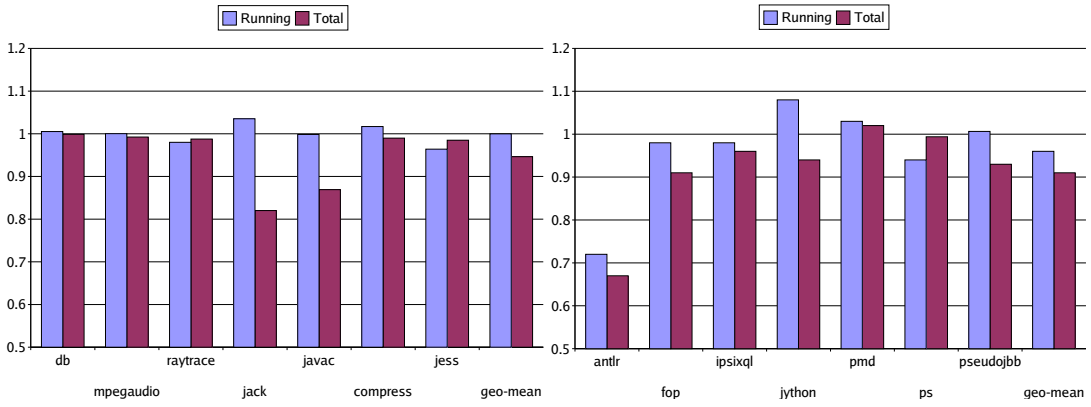
For the *Opt* scenario, shown in Figure 9, we obtain an even better result. We improve total time on average by 6% with large reductions in total time for jack (18%) and javac (13%) with no change in running time. And, we achieve nice reductions on our test suite with average reductions in running time of 4% and total time of 9%. We get significant reductions in total and running time for antlr (28% and 33%) and we get good reductions in total time for fop (9%), pseudojbb (7%), and jython (6%).




(a) Performance on SPECjvm98

(b) Performance on DaCapo + JBB

**Figure 8. Adaptive scenario, *Adapt*, tuned for balance on PPC (Adapt:PPC)**




(a) Performance on SPECjvm98

(b) Performance on DaCapo + JBB

**Figure 9. Optimizing scenario, *Opt*, tuned for balance on PPC (Opt:PPC)**

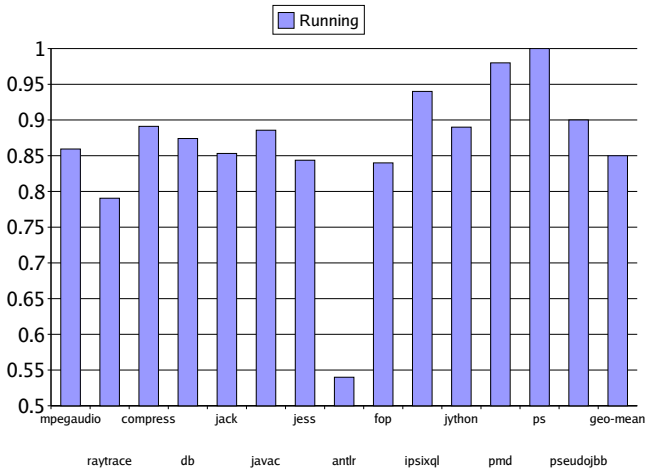
### 6.5 Tuned for Running time of each benchmark

Although, the main focus of this paper has been in developing an off-line heuristic to reducing the total exectution time of programs, there may be occasionally long-running programs where it is worth while expending effort to find the best running time as compilation time is relatively insignificant.

Figure 10 shows results for the best performing heuristic that the genetic algorithm found by tuning the heuristic for each program. For all the SPECjvm98 benchmarks, we are able to achieve more than 10% improvement in running time (with 4 out of 7 reduced by almost 15%). The results for DaCapo+JBB benchmarks were more varied. We achieve a large reduction of 46% for antlr and for fop, jython, and pseudojbb we achieve at least 10% reductions or more. Interestingly, for ps we are not able to find any signficant reductions. On average, we can achieve a 15% improvement in running time for all programs.

### 6.6 Summary of Results

Table 5 shows the average reductions for running times and total times for both our benchmark suites. Since our genetic algorithm was tuned over SPECjvm98, as to be expected we always outperform the default heuristic giving a 6% and 17% reduction in total execution time on the PowerPC and Intel platform respectively. For the DaCapo+JBB benchmark suite, we almost always are able to outperform the default heuristic. We do incur a degradation for running time, when tuning for total time, but this is expected and this leads to a 37% reduction in total execution time for the Intel platform. On the PowerPC architecture we




(a) Performance of Optimizing Scenario on X86

**Figure 10. Running time reduction when tuning for each program in turn**

| Compilation Scenarios | SPECjvm98 | | DaCapo+JBB | |
|---|---|---|---|---|
| | Running | Total | Running | Total |
| Adapt | 6% | 3% | 0% | 29% |
| Opt:Bal | 4% | 16% | 3% | 26% |
| Opt:Tot | 1% | 17% | -4% | 37% |
| Adapt (PPC) | 5% | 1% | -1% | 6% |
| Opt:Bal (PPC) | 0% | 6% | 4% | 9% |

**Table 5. Average Performance of Genetically Tuned Heuristic**

incur a slight running time degradation for Adapt of 1% which gives a good reduction in total execution time. These results suggest that the Jikes RVM heuristic has been primarily tuned for adaptive compilation for the SPECjvm98 benchmark suite on the PowerPC and that our approach is able to adapt to new benchmarks, platforms and compilation scenarios.

## 7 Related Work

Here we critically review the most relevant papers in inlining and machine learning based compilation.

**Inlining** Arnold et al. [3] formulate the size/speed tradeoffs of inlining as a Knapsack problem. They do not measure total execution-time, but instead focus on code size and the running time of the program. This work is however a theoretical limit study as they assume global knowledge of the program when making an inlining decison, information not available to a dynamic compiler. They show there is the potential to achieve significant speed up in running time (on average 25%) over no inlining with modest limits on code expansion (up to 10%). They also show that in several cases performance degradation of a program's running time can occur due to overly aggressive inlining. A central obstacle with the formulation of inlining as a Knapsack problem to dynamically compiled languages such as Java, is its requirement on a global view of the entire program. However, this is typically not the case when compiling Java programs because compilation (and inlining decisions) occurs when a method is about to be invoked. Therefore, heuristics are used to make inlining decisions for Java.

Hazelwood et al.[9] describe a technique of using context sensitive information at each call site to control inlining decisions enabling 10% reductions in code space at the expense of a degradation in total execution time. 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 senstivity, but there is no clear winner among them. This technique is orthogonal to our tuning process, in fact, this technique actually introduces an additional heuristic used for inlining which may benefit for our automatic tuning process.

Dean et al.[7] present a technique for measuring 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 ef-

fects of optimizations applied to the body of the inlined routine . 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.

Cooper et al.[5] actually show a degradation in performance of numerically intense Fortran benchmarks from inlining. Inlining of certain critical calls lead to poorer subsequent analysis and less effective instruction scheduling which resulted in an increased number of floating-point stalls. This study further emphasizes our point that more intelligent inlining is required and tuning heuristics through empirical search as opposed to imprecise modeling is beneficial.

Leupers et al. [10] 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 targetted to 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 amortised over the products shipped but is not practical for non-embedded applications.

**Machine Learning** Stephenson *et al.* [15] 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.

However, these two pre-existing heuristics were not well implemented. The authors even admit that turning off data prefetching completely is preferable and reduces many of their significant gains. For the third optimization, register allocation, they were only able to achieve on average a 2% increase over the manually tuned heuristic.

Cooper *et al.* [6] 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. [12] use a classifier based on decision tree learning to determine which loops to unroll. They looked at the performance of compiling Fortran programs from the SPEC benchmark suite using g77 for two different architectures, an UltraSPARC and an IA64. They showed an improvement over the hand-tuned heuristic of 3% and 2.7% over g77's unrolling strategy on the IA64 and UltraSPARC, respectively.

## 8 Conclusions

Inlining is an important optimization for many programming languages. It has been well-studied and as a consequence well optimized within Jikes RVM, a high-performance Java optimizing compiler. However, optimizing inlining heuristics is a time-consuming and difficult process. We describe a technique using a genetic algorithm to tune inlining heuristics that automates the process of developing optimizing heuristics that improves on program performance.

Using a genetic algorithm, we can obtain heuristics that significantly outperform the existing hand-tuned heuristic. On the set of benchmarks we learned our heuristics on, we are able to achieve an average reduction of 17% in total running time on an Intel machine and a 6% reduction on a PowerPC. More importantly, on an "unseen" set of benchmarks we can reduce the average total running time by an impressive 37% on an Intel machine and by 7% on a PowerPC. Furthermore, when specializing a heuristic for each application, we can achieve an average reduction of 15% in running time over the default heuristic. Our results not only show that automatic optimization is better than manual compiler writer tuning, but that different heuristics are required for different compilation scenarios and/or architectures. We conclude that genetic algorithms are successful at finding good inlining heuristics and shows promise for its application in tuning other compiler heuristics.

## References


[1] 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 Journal*, 39(1):211–238, Feb. 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 47–65, Minneapolis, MN, Oct. 2000. ACM Press.

[3] 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.

[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] K. D. Cooper, M. W. Hall, and L. Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[6] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, Atlanta, Georgia, July 1999. ACM Press.

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

[8] J. G. Forum. Making java work for high-end computing. In *Supercomputing*, 1998.

[9] 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.

[10] 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.

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

[12] 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.

[13] D. Project. DaCapo Benchmarks. http://www-ali.cs.umass.edu/DaCapo/gcbm.html, 2004.

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

[15] 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.