Automated Just-In-Time Compiler Tuning

Kenneth Hoste Andy Georges Lieven Eeckhout
Ghent University, Belgium
{kehoste, ageorges, leeckhou}@elis.ugent.be

Abstract

Managed runtime systems, such as a Java virtual machine (JVM), are complex pieces of software with many interacting components. The Just-In-Time (JIT) compiler is at the core of the virtual machine, however, tuning the compiler for optimum performance is a challenging task. There are (i) many compiler optimizations and options, (ii) there may be multiple optimization levels (e.g., `-O0`, `-O1`, `-O2`), each with a specific optimization plan consisting of a collection of optimizations, (iii) the Adaptive Optimization System (AOS) that decides which method to optimize to which optimization level requires fine-tuning, and (iv) the effectiveness of the optimizations depends on the application as well as on the hardware platform. Current practice is to manually tune the JIT compiler which is both tedious and very time-consuming, and in addition may lead to suboptimal performance.

This paper proposes automated tuning of the JIT compiler through multi-objective evolutionary search. The proposed framework (i) identifies optimization plans that are Pareto-optimal in terms of compilation time and code quality, (ii) assigns these plans to optimization levels, and (iii) fine-tunes the AOS accordingly. The key benefit of our framework is that it automates the entire exploration process, which enables tuning the JIT compiler for a given hardware platform and/or application at very low cost.

By automatically tuning Jikes RVM using our framework for average performance across the DaCapo and SPECjvm98 benchmark suites, we achieve similar performance to the hand-tuned default Jikes RVM. When optimizing the JIT compiler for individual benchmarks, we achieve statistically significant speedups for most benchmarks, up to 40% for start-up and up to 19% for steady-state performance. We also show that tuning the JIT compiler for a new hardware platform can yield significantly better performance compared to using a JIT compiler that was tuned for another platform.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Experimentation, Measurement, Performance

Keywords Java Virtual Machine (JVM), Just-In-Time (JIT) compiler, compiler tuning, evolutionary search, machine learning

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.

1. Introduction

One of the key advantages of managed programming languages, such as Java, is that programs are compiled to an intermediate machine-independent level, called bytecode, enabling cross-platform portability. However, this requires a process virtual machine—a Java virtual machine or JVM for short—to translate bytecode to executable code. Modern JVMs tend to follow a mixed-mode execution scheme in which application methods are first interpreted, or compiled with a baseline non-optimizing compiler. If a method is sufficiently *hot*, i.e., is executed frequently, it will likely be a candidate for (re)compilation by the optimizing JIT compiler. In this paper, we refer to a set of optimizations used together during the (re)compilation of a method as an *optimization plan*. Modern JVMs [3, 19, 21] employ multiple *optimization levels* (e.g., `-O0`, `-O1` and `-O2`), in which each level comprises a successively more aggressive optimization plan. In other words, more aggressive optimizations are performed on more frequently executed code: higher optimization levels result in longer compilation times, yet they supposedly yield better code, thereby further speeding up the execution of the hot methods.

Tuning the VM's JIT compiler is a challenging task for a number of reasons. For one, to ensure good performance, the VM developer has to carefully tune each of the optimization levels, choosing the right optimizations at each level and tweaking their settings and controls. This is far from trivial because of the large number of available optimizations and their complex interactions. Second, the Adaptive Optimization System (AOS), i.e., the engine that decides which methods to optimize to which optimization level, needs to be fine-tuned. This is non-trivial as well because the optimum AOS configuration is highly dependent on the compilation plans at each optimization level and it is crucial to take full advantage of the available optimization levels. Third, this tuning process needs to be done for every possible optimization target of interest. In particular, the optimal VM configuration

may be specific to a particular hardware platform because different hardware platforms come with different memory hierarchies, microarchitecture, etc. which requires the JIT compiler to be tuned differently. Different applications may need the JIT compiler to be tuned differently as well. For example, servers often run a single application or a limited number of applications, such as middle-ware or business applications, over and over again. As such, it makes sense to tune the VM for a particular application or set of applications.

Current practice is to manually tune the JIT compiler. Arnold et al. [4] and Ishizaki et al. [16] describe such a manual process for the Jikes RVM and the IBM JDK production VM, respectively. This process is both tedious, time-consuming and costly, and may lead to sub-optimal performance. Moreover, tuning needs to be done for every new processor on the market as well as for different applications and application domains.

This paper proposes automated JIT compiler tuning. This is done in two steps. The first step identifies optimization plans that are Pareto-optimal in terms of compilation time and code quality—a Pareto-optimal plan is defined such that there exists no optimization plan that performs better on both compilation time *and* code quality. We use a multi-objective evolutionary search algorithm to efficiently search the large optimization space: starting from a set of randomly generated optimization plans, we let the algorithm evolve until it converges on a set of Pareto-optimal plans. We subsequently retain a limited number of optimization plans that cover the Pareto frontier well. The second major step is to search for the optimum JIT compiler. This involves assigning Pareto-optimal compilation plans to optimization levels (`-O0`, `-O1` and `-O2`), and fine-tuning the AOS. Again, we use evolutionary search for doing so. The end result is a VM that is optimized for the optimization target(s) of interest, i.e., for a given hardware platform and/or application domain.

Our experimental results using the Jikes RVM, the DaCapo and SPECjvm98 benchmarks, and four different hardware platforms demonstrate the value of the proposed framework. The key experimental results from this paper are as follows:

- We show that the framework succeeds in automatically tuning a modern JIT compiler. We report similar average performance compared to the manually tuned Jikes RVM.
- Tuning for a particular benchmark and a particular hardware platform yields statistically significant speedups of up to 19% for steady-state and 40% for start-up performance.
- Tuning a VM for a new hardware platform yields significantly better performance compared to a VM that was tuned for another platform.

Overall, this paper makes the following key contributions:

- We are the first to propose a framework for automatically tuning a JIT compiler with multiple optimization levels for optimum performance. The key benefit is that the exploration is fully automated and enables tuning the JIT compiler for a given hardware platform and/or (set of) application(s) at very low cost.
- We provide empirical evidence that substantial performance gains can be obtained by tuning the JIT compiler for a particular hardware platform and/or application.
- We make the case that tuning a dynamic (JIT) compiler is much more complicated than tuning a static compiler because of the tight interaction between the optimization plans and levels and the AOS. This insight motivated us to propose a two-step process in which we first identify Pareto-optimal optimization plans, and subsequently assign plans to levels and fine-tune the AOS.

Although this paper uses the Jikes RVM for driving the experiments, we strongly believe that the overall framework and conclusion is applicable to other Java virtual machines. Moreover, similar JIT compiler tuning can be applied on other process virtual machines, such as the Common Language Runtime (CLR) of the Microsoft's .NET initiative.

The remainder of this paper is organized as follows. Section 2 gives a detailed description of the organization of a modern JVM, namely Jikes RVM. In Section 3, we present our JIT optimization space exploration algorithm. In Section 4, we describe our experimental setup, and we present the results in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. Java Virtual Machine

Before presenting the proposed JIT compiler optimization framework, we first briefly describe the organization of a modern Java virtual machine, namely Jikes RVM [3]. This will enable us to better understand the complexity of JIT compiler tuning.

Optimization plans and levels. Jikes RVM is a compilation-only VM. Methods are initially compiled using a fast but non-optimizing baseline compiler that generates relatively inefficient machine code. To improve performance, Jikes RVM employs an JIT optimization strategy for optimizing hot methods using three optimization levels (`-O0`, `-O1` , and `-O2`). We refer to the baseline compilation level as `base`.

Each optimization level `-On` is defined by an optimization plan $P_{\texttt{On}}$ that enumerates the optimizations at that level along with several values that further steer their use. In the default Jikes RVM configuration, optimization plans for higher levels include the optimizations for the lower levels. Each optimization level also has a corresponding *aggressiveness* assigned to it that influences the use of various optimizations, e.g., more copy propagation passes are done at higher opti-

	base	-O0	-O1	-O2
compilation rate (bs/ms)	909.46	39.53	18.48	17.28
speedup vs. base	1.0	4.03	5.88	5.93

Table 1: Default compiler DNA values for Jikes RVM.

mization levels. In Jikes RVM (version 3.0.1), there are 33 boolean options available, each of which turns an optimization on or off, and 10 value options that control the optimizations[1]. Thus, per optimization plan, we have 2^{33} possible combinations of boolean flags and a space spanned by eight positive integer values and two positive floating-point values. This results in a huge search space.

Compiler DNA. A compilation plan is characterized using two metrics: the compilation rate (i.e., bytecodes compiled per millisecond (bc/ms), and the improvement in code quality (i.e., speedup in execution time over `base`). Combined, these two metrics are referred to as the *compiler DNA* associated with the optimization plan.

The compiler DNA for each optimization plan/level in Jikes RVM is measured as follows. The compilation rate is obtained by compiling *all* methods at the specified optimization level upon first execution. The speedup is the ratio between the execution time obtained by executing the optimized code and the execution time for a VM using the `base` compiler only. The DNA in Jikes RVM for x86, see Table 1, was computed on an LS41 type 7972 blade, equipped with an AMD Opteron 8218 with 4MB L2 cache and 4GB RAM, using the SPECjvm98 benchmarks[2].

Sample-based JIT optimization. Jikes RVM uses OS-timer triggered sampling to identify hot methods. When the timer fires, the method on top of the stack is sampled the moment a yield point[3] is reached [4, 5]. When sufficient samples have been gathered for a method, the VM uses the AOS to decide whether or not to optimize the method to a particular optimization level.

Adaptive Optimization System. The AOS decides whether or not to optimize a method, and if so, to which optimization level the method should be optimized. There are five value options in total that control the AOS. There are three positive integer values, and two positive floating-point values in total, again, a large space to explore. The AOS parameters control when the engine finds a method to be hot enough to be considered for optimization to a higher level. The AOS uses the compiler DNA to make a trade-off in compilation cost (i.e., how long does it take to optimize the method at a given optimization level?) and code quality (i.e., how much faster will the code run once optimized?).

[1] These are the options we have used in the exploration. There are other options we did not use because they are either unstable, not meant to be changed from outside the VM or can activate options that result in breaking the Java language specification.

[2] The Jikes RVM compiler DNA for the PowerPC platform specifies different values.

[3] A yield point in Jikes RVM is a point during the execution where the scheduler can safely switch threads. It is placed at the beginning and the end of methods and at loop back-edges.

3. JIT Compiler Tuning

We now present our framework for automatically optimizing a JIT compiler. This includes identifying the compilation plans, optimization levels and AOS settings. Before describing the overall framework in great detail, we first motivate the need for a two-step process.

3.1 Why a two-step process?

As mentioned earlier in the introduction, optimizing a dynamic compiler is substantially more complicated than optimizing a static compiler because of the tight interaction between optimization plans and levels, and the AOS settings. For example, including a compiler optimization at one level changes the compilation rate versus code quality trade-off, which in its turn changes which methods are optimized to which optimization level. This leads to complex interactions which severely complicate the search process. Our initial approach to this problem was to use an evolutionary algorithm to optimize the compilation plans, plan-to-level assignments, the number of optimization levels, and the AOS settings in a single go. In fact, we used the previously proposed COLE approach [15] which was developed for a static compiler, and naively applied it to a dynamic compiler. However, we encountered three significant problems. First, the automatically derived JIT compiler did not perform better (and for many benchmarks significantly worse) than the manually tuned Jikes RVM. Second, the search process took extremely long to converge. Third, expressing the optimization problem in a format that could be handled by COLE's evolutionary search algorithm was non-trivial, e.g., it is unclear how to sensibly define crossover across two JIT compiler settings with a different number of optimization levels. This motivated us to come up with a two-step process in which we first focus on code quality versus compilation rate while excluding dynamic compilation and GC activity, and subsequently assign plans to levels and optimize the AOS settings while considering dynamic compilation and GC activity. The two-step process enables a higher performance JIT compiler to be derived in a shorter amount of time.

3.2 Pareto-optimal optimization plans

The goal of the first step is to identify optimization plans that are Pareto-optimal in terms of compilation time and code quality. Figure 1 shows an illustrative example of a Pareto frontier in the dual-objective search space, namely compilation rate (i.e., number of bytecodes compiled per unit of time) versus speedup (i.e., performance improvement compared to non-optimized code). A compilation plan is Pareto-optimal if there is no other plan that performs better both in terms of compilation rate and speedup. When constructing the Pareto frontier, we consider a setup in which we first

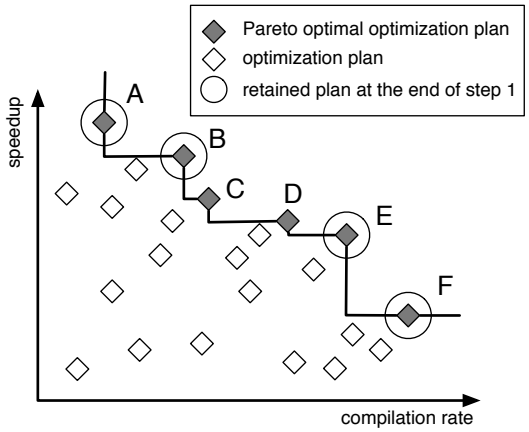


Figure 1: An example of a Pareto frontier in our dual-objective exploration space. The circled plans are those retained at the end of the first step to bootstrap the second step.

compile all the code according to the optimization plan and subsequently execute the optimized code—we do not consider JIT compilation (for now) and consider a large heap size (8 times the minimum heap size) to minimize GC activity. This is to understand the basic trade-off in code quality versus optimization overhead.

For identifying the Pareto frontier, we use the SPEA2 multi-objective evolutionary search algorithm [27], which was also used in the COLE framework [15]. In our implementation, the algorithm starts with a *generation* of 25 compilation plans: one plan with all compilations turned on, one plan with all optimizations turned off, and 23 randomly generated compilation plans. Each of these plans are evaluated in terms of compilation rate and speedup. The best compilation plans seen so far are retained in an *archive*, which contains the Pareto-optimal plans. The next generation is formed by probabilistically mutating plans and combining them using crossover. In our setup, we use mutation to construct $1/10$ of the plans in the next generation with a mutation rate of 25%, and crossover for $9/10$ of the plans with a crossover rate of 25%. After evaluating this new generation, we retain the Pareto-optimal entities in the new archive. This iterative process is repeated until convergence, i.e., until there is no further improvement in the Pareto frontier. The final Pareto frontier contains all the Pareto-optimal optimization plans ever seen during the exploration.

3.3 Limiting the number of Pareto-optimal optimization plans

The end result of the multi-objective evolutionary search as described above is a fairly large set of Pareto-optimal optimization plans; in our experiments, we obtained up to 80 Pareto-optimal plans. From this set, we select a subset such that the Pareto frontier is covered well. We found this Pareto frontier reduction procedure to be an important step in the overall JIT exploration in order to limit the total exploration time.

The rationale behind the Pareto frontier reduction procedure is to prefer optimization plans that result in high code quality at roughly the same compilation rate, and compile bytecode faster while attaining roughly the same speedup. We therefore use an iterative selection algorithm. In the first iteration, we pick the two adjacent plans on the Pareto frontier that lie closest to each other along the X axis. We drop the plan that scores worst along the Y axis. We then select the two plans that lie closest to each other along the Y axis, and drop the one that scores worst along the X axis. This iterative process stops when the number of retained plans drops below a given number. We limit the number of retained Pareto-optimal compilation plans to 8.

In our running example, see Figure 1, this means we first select the pair (B,C) because they lie closest on the X axis and drop C. The next pair is (D, E) because they lie closest on the Y-axis and we only retain E. After two iterations, the list of retained optimization plans equals $\{$A, B, E, F$\}$.

3.4 JIT compiler tuning

The second step in the proposed JIT compiler tuning framework is to (i) assign the Pareto-optimal optimization plans to optimization levels (`-O0`, `-O1` and `-O2`), and (ii) optimize the JIT AOS accordingly. In contrast to the first step, we now consider adaptive JIT compilation, i.e., the JIT compiler optimizes the most frequently executed methods at run time, and we consider heap sizes that introduce GC activity in order to achieve representative performance numbers. In other words, compilation and optimization time as well as GC time become part of the overall execution time.

Assigning compilation plans to optimization levels is fairly straightforward: given the limited number of retained Pareto-optimal optimization plans we can easily consider all possible assignments of plans to levels. In our setup, this means we need to assign 8 optimization plans to 1 through 3 optimization levels. There are 92 possible assignments. We use an evolutionary search algorithm to identify the best AOS settings.

4. Experimental Setup

In this section, we describe our experimental setup in terms of the benchmarks, the hardware platforms, the Jikes RVM version, and the data analysis method used.

4.1 Benchmarks

Table 2 shows the benchmarks used in this study. We use the SPECjvm98 benchmarks [23] (top seven rows), as well as nine DaCapo benchmarks [7] (bottom nine rows). SPECjvm98 is a client-side Java benchmark suite consisting of seven benchmarks. We run all SPECjvm98 benchmarks with the largest input set (`-s100`). The DaCapo benchmark suite is an open-source benchmark suite; we use release version 2006-10-MR2. We use the nine benchmarks that exe-

benchmark	description	min heap size (MB)
compress	file compression	24
jess	puzzle solving	16
db	database	32
javac	Java compiler	32
mpegaudio	MPEG decompression	16
mtrt	raytracing	24
jack	parsing	24
antlr	parsing	32
bloat	Java bytecode optimization	56
fop	PDF generation from XSL-FO	56
hsqldb	database	176
jython	Python interpreter	72
luindex	document indexing	32
lusearch	document search	32
pmd	Java class analysis	64
xalan	XML to HTML transformer	40

Table 2: SPECjvm98 (top seven) and DaCapo (bottom nine) benchmarks considered in this paper.

cute properly on the 3.0.1 version of Jikes RVM. We use the default (medium size) input set for the DaCapo benchmarks unless mentioned otherwise.

4.2 Hardware platforms

We use four different hardware platforms in this study:

- an AMD Opteron 242 clocked at 1.6GHz with 1MB L2 cache and 4GB RAM running Linux 2.6.9;
- an Intel Pentium 4 clocked at 3GHz with 1M L2 cache and 1.5GB RAM running Linux 2.6.19;
- an Intel Core 2 based Xeon L5420 clocked at 2.5GHz with 6MB L2 cache and 16GB RAM running Linux 2.6.18; and
- an Intel Core i7 920 based machine clocked at 2.6GHz with 256KB L2, 8MB L3 and 12GB RAM running Linux 2.6.27.

4.3 Jikes RVM

We use Jikes RVM version 3.0.1, released on November 18th, 2008. We patched Jikes RVM such that optimizations can be set on a per-optimization level basis at the command line. The virtual machine was built using the *production* profile, which uses the GenMS garbage collector and compiles the VM methods using the optimizing compiler with the default P_{02} optimization plan. During the first step of the exploration algorithm, we use a heap size that is 8 times the minimum size required to run the benchmark; this is to eliminate the effect of garbage collection, as mentioned earlier. We do vary the heap size (i.e., $2\times$, $4\times$, and $8\times$ the minimum heap size) during the second step and during evaluation, following current practice [7].

Plan	Compilation rate	Speedup (over `base`)
`-O0`	53.12	1.86
`-O1`	21.84	2.14
`-O2`	20.81	2.13
A	59.70	1.77
B	57.62	1.86
C	50.86	1.89
D	41.07	2.00
E	37.42	2.02
F	28.70	2.05
G	25.90	2.08
H	19.11	2.13

Table 3: Compilation rates and speedups over `base` on the Intel Core 2 for the compilation plans used by default in Jikes RVM (top rows), and the compilations plans obtained through our exploration (bottom rows).

4.4 Statistically rigorous performance evaluation

To deal with the non-determinism that is due to timer-based sampling and adaptive optimization in Jikes RVM, we use both multiple VM invocations and multiple benchmark iterations per VM invocation in our experiments, following the statistically rigorous performance evaluation methodology proposed by Georges et al. [13]. When reporting start-up performance we consider the average execution time for the first benchmark iteration across 20 VM invocations. When reporting steady-state performance we consider the arithmetic mean across the final 5 out of 15 benchmark iterations across 20 VM invocations. We also report 95% confidence intervals which are indicated through error bars in the graphs.

5. Results

We now evaluate the proposed JIT compiler tuning framework. We consider three cases: (i) tuning for average performance across all benchmarks, (ii) tuning for a particular benchmark, and (iii) tuning for a specific hardware platform. We consider experimental setups both with and without cross-validation. Finally, we discuss the exploration time.

5.1 Tuning for a benchmark suite

For now, we use all the benchmarks from the SPECjvm98 and DaCapo suites, and aim at finding a JIT compiler setting that performs well on average across all of the benchmarks. Our goal is to demonstrate that automated JIT compiler tuning performs at least as well as a manually tuned JIT compiler. This exploration was conducted on the Intel Core 2 platform.

Pareto-optimal optimization plans. Table 3 lists the three default compilation levels as well as the compilation plans we obtained from the first step in our exploration process in terms of compilation rate and speedup (code quality). The

automatically derived Pareto-optimal compilation plans are comparable to the manually tuned compilation plans in default Jikes RVM, and are well spread in terms of compilation rate and code quality.

Optimum JIT compiler. The second step is to identify optimum plan-to-level assignments and AOS settings. We denote the JIT compiler that optimizes start-up performance as C_{ST}; the JIT compiler that optimizes steady-state performance is denoted as C_{SS}. These settings are shown in Table 4; interestingly, the optimum start-up JIT compiler C_{ST} has three levels with plans E, C and A, whereas the optimum steady-state JIT compiler C_{SS} has only two levels with plans E and A. We found the automatically tuned JIT compiler to achieve significantly better performance than the manually tuned Jikes RVM for a couple benchmarks, e.g., `mtrt` (30% for start-up and 7% for steady-state), `hsqldb` (10% for start-up) and `bloat` (3% for steady-state). For some benchmarks, we observe slightly worse performance, e.g., `lusearch` and `xalan` for steady-state; performance degradation is limited to 3% to 4% though. However, for the majority of the benchmarks, we do not observe statistically significantly better or worse performance. Overall, the end conclusion is that *automated JIT compiler tuning is feasible and achieves similar performance compared to a manually tuned JIT compiler*.

5.2 Cross-validation experiment

The evaluation described so far assumed that the JIT compiler was tuned and evaluated using the same set of benchmarks, namely DaCapo and SPECjvm98. Even more relevant is to study whether one could tune the JIT compiler with one set of benchmarks and then achieve good performance for other benchmarks. We now employ such a cross-validation setup: we tune the JIT compiler using the DaCapo benchmark suite and then evaluate the tuned JIT compiler using the SPECjvm98 benchmark suite, and vice versa. Figure 2 shows the results of this cross-validation experiment along with the results of a non cross-validation experiment (i.e., the JIT compiler is tuned and evaluated using the same set of benchmarks), which serves as a point of reference. For SPECjvm98 (top row), we observe that the automatically tuned JIT compiler achieves good performance even in a cross-validation experiment (compare Figure 2(a) to the non cross-validation experiment in Figure 2(b)). The automatically tuned JIT compiler achieves substantial speedups for mtrt and compress. We observe a slowdown for mpegaudio in the cross-validation setup. The performance picture is mixed for the DaCapo benchmark suite (bottom row): when tuned for SPECjvm98, the JIT compiler performs worse for some of the DaCapo benchmarks, see for example bloat, jython, lusearch and pmd. For the other benchmarks, we observe similar (or similarly good, see hsqldb) performance under cross-validation. The reason for the different performance picture for DaCapo compared to SPECjvm98 is due to the significant differences in workload characteristics between DaCapo and SPECjvm98: Blackburn et al. [7] demonstrate that DaCapo shows more complex code, has richer object behaviors, and has more demanding memory system requirements. This result motivates the need for representative benchmarks when (automatically) tuning a JIT compiler—this is a general concern for feedback-loop based optimization and tuning.

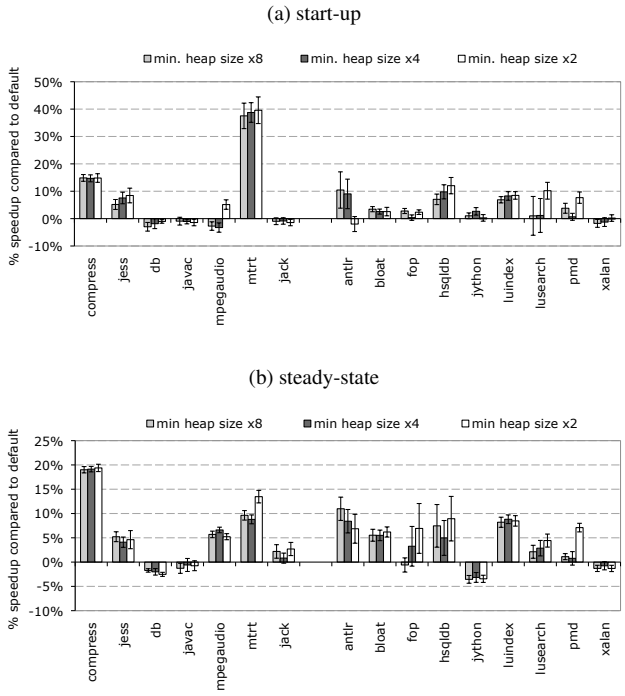


Figure 3: Speedup on the Intel Core 2 compared to default Jikes RVM for start-up and steady-state performance when tuning the JIT compiler for optimum performance on a per-benchmark basis.

5.3 Tuning for a single benchmark

An important benefit from automated JIT compiler tuning is that it enables the optimization for specific applications as well as for specific hardware platforms at very low cost, given that the tuning process is completely automated. In this section, we discuss the results we obtain when we tune the JIT compiler for a specific benchmark; we discuss the case in which we tune for a specific hardware platform later.

Figure 3 shows the speedup on the Core 2 platform when comparing the best Pareto-optimal configuration tuned per benchmark for (a) start-up and (b) steady-state performance against the default Jikes RVM. The automated exploration yields JIT compilers that outperform the default Jikes RVM for a good portion of the benchmarks, and up to 40% for start-up and up to 19% for steady-state.

5.4 Cross-input validation

In the previous section, we considered the same benchmark inputs when tuning the JIT compiler as during evaluation.

	default	C_{ST}	C_{SS}
number of levels	3	3	2
level 0	P_{00}	plan E	plan E
level 1	P_{01}	plan C	plan A
level 2	P_{02}	plan A	–
Number of clocks ticks after which call graph decays	100	52	26
Call graph decay rate	1.10	1.10	1.10
Call graph update frequency in timer ticks	20	3	4
Initial edge weight in call graph	3	3	3
Percentage of edges that mark hotness	0.01	0.0136	0.0098

Table 4: The JIT compiler configurations that are optimal in terms of startup (C_{ST}) and in terms of steady-state (C_{SS}).

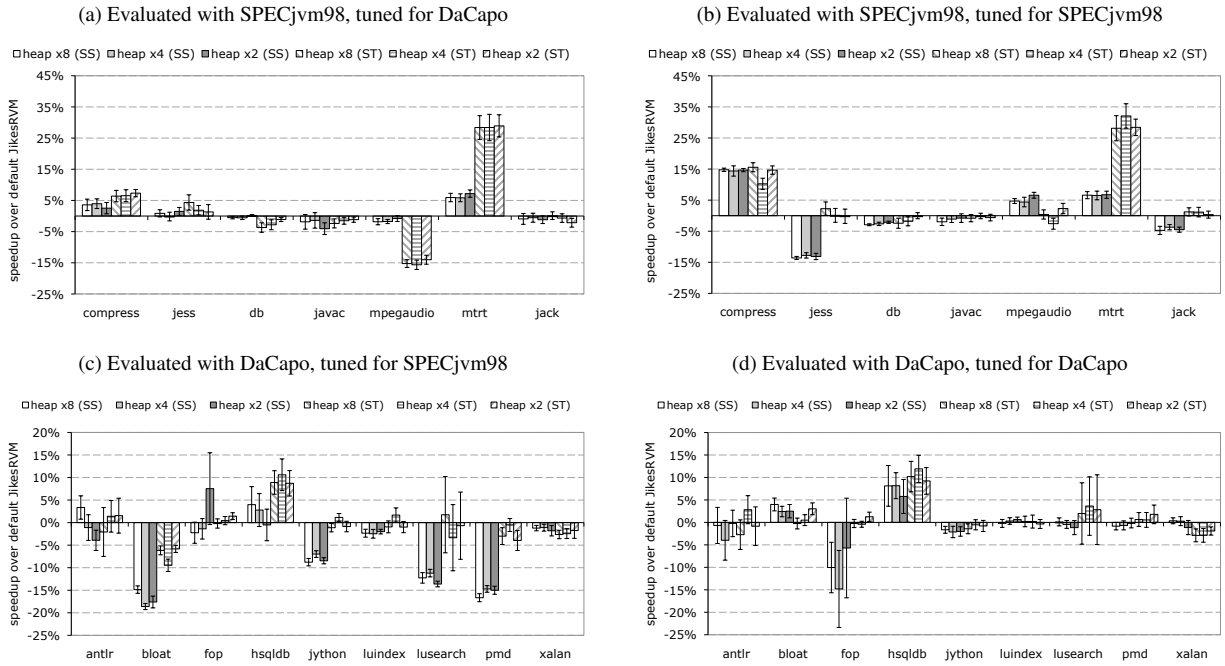


Figure 2: Per-benchmark performance speedups on the Intel Core 2 compared to default Jikes RVM when tuning Jikes RVM in a cross-validation setup (left column: (a) + (c)), and a non cross-validation setup (right column: (b) + (d)). These graphs show results for both startup (ST) and steady-state (SS) performance, across three heap sizes.

Figure 4 reports performance results when considering a different input during the tuning process and evaluation, i.e., we now consider a cross-input validation setup. We limit ourselves to the DaCapo benchmarks in this experiment: we use the medium inputs during JIT compiler tuning and use the large inputs during evaluation. Two DaCapo benchmarks are excluded, namely `fop` and `luindex`, because the medium input is equal to the large input. We do not consider SPECjvm98 here because of lack of inputs: the `-s1` and `-s10` inputs are too small and only stress virtual machine startup performance and do not stress code quality [12].

Comparing Figures 3 and 4, we observe roughly the same speedup for the medium inputs (Figure 3) as for the large input (Figure 4) for some benchmarks, e.g., hsqldb. For other benchmarks, we observe a slight performance drop, e.g., bloat. This motivates the need for representative inputs when tuning a JIT compiler for a particular application — an input that yields substantially different program behavior than the input used during the tuning process may result in suboptimal performance. As mentioned before, this is a general concern for feedback-loop based optimization and tuning.

5.5 Tuning for a specific hardware platform

We now explore the potential performance benefit by tuning the JIT compiler for a specific hardware platform. In this case study, we examine the effects of tuning for a particular platform using two benchmarks: (i) `mtrt`, and (ii)

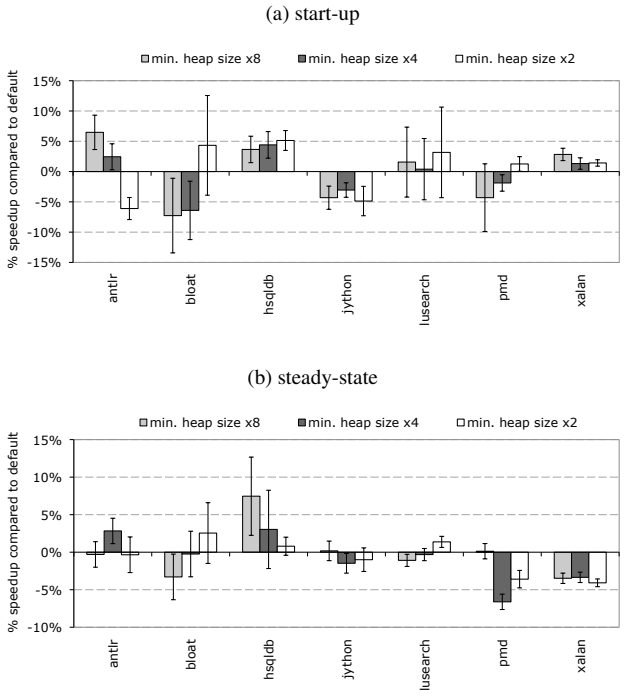


Figure 4: Per-benchmark start-up and steady-state speedup through a cross-input validation experiment.

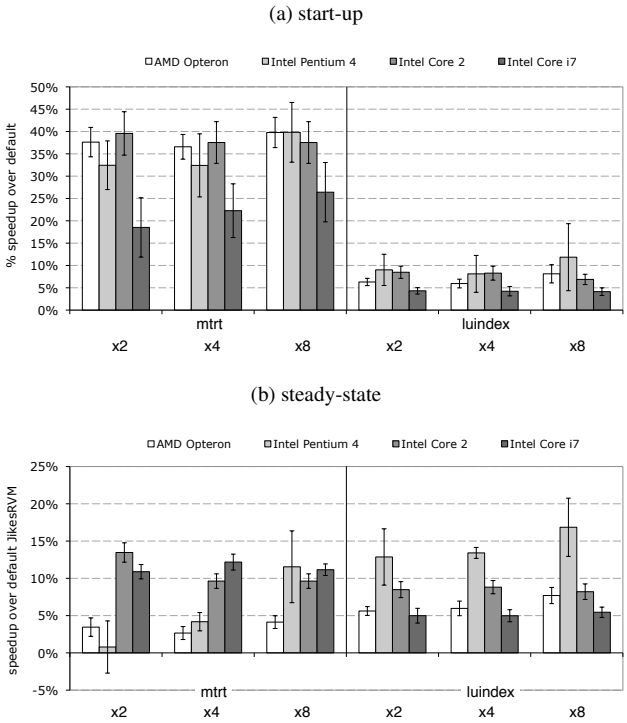


Figure 5: This graph shows speedup numbers across different heap sizes on all hardware platforms for `mtrt` and `luindex` for (a) start-up and (b) steady-state performance.

`luindex`. During the benchmark suite wide exploration on the Intel Core 2 (see Figure 2), the optimum JIT compiler did very well for `mtrt`, yet it failed to improve performance for `luindex`. In the per-benchmark exploration, `mtrt` improves further, while `luindex` gains 9% in both start-up performance and in steady-state performance on the Core 2, see Figure 3. Thus, these two benchmarks make for two excellent cases for examining the effect of exploring benchmark-specific tuning across different hardware platforms.

Per-platform tuning. Our first experiment tunes the JIT compiler for a specific hardware platform and compares performance against the default JIT compiler (which was manually tuned for another hardware platform). The results of this hardware-specific exploration are shown in Figure 5 for (a) start-up and (b) steady-state performance. There is significant benefit in start-up performance for `mtrt`: performance speedups range from 19% to 40%. For `luindex` we observe performance benefits in the 4% to 12% range. For steady-state performance, we observe substantial performance benefits for both `mtrt` and `luindex`: performance improves by up to 13% for `mtrt` and up to 17% for `luindex`. These examples make the case that significant speedups can be obtained from tuning a JIT compiler for a specific hardware platform.

This result is further illustrated in Figure 6 which shows the per-platform Pareto-optimal optimization plans in terms of compilation rate versus performance speedup over `base`. There are two observations we can immediately draw from

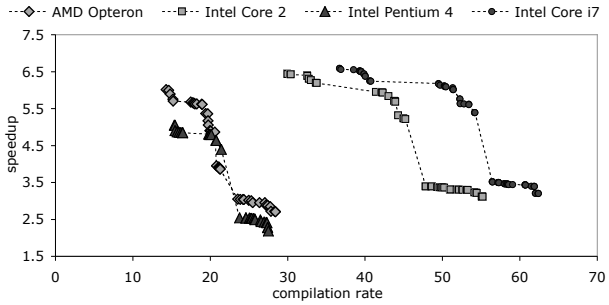
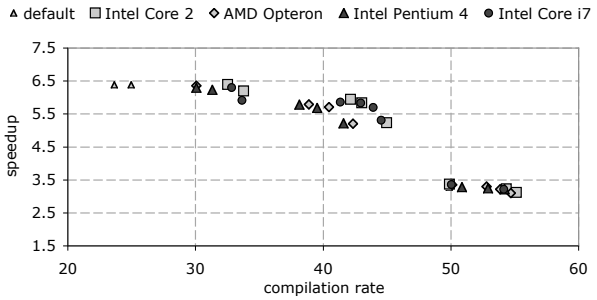


Figure 6: The Pareto frontiers for the optimization plans tuned for `mtrt` on each of the platforms in our experimental setup.

this graph. First, the frontier shifts to the upper right for more recent platforms. As a consequence, if one tunes the compiler DNA on an older platform, the cost for optimization is over-estimated and the potential benefit the optimization reaps is under-estimated. This results in optimizing either later—more samples are required to overcome the decision threshold—or optimizing to a lower level. Conversely, if the VM is tuned for a more recent platform, the VM might optimize too soon and/or too much, potentially offsetting the gain the adaptive framework might bring. Second, we see that on each platform the Pareto frontier is well spread across

(a) Optimization plans



(b) Tuned JIT compilers

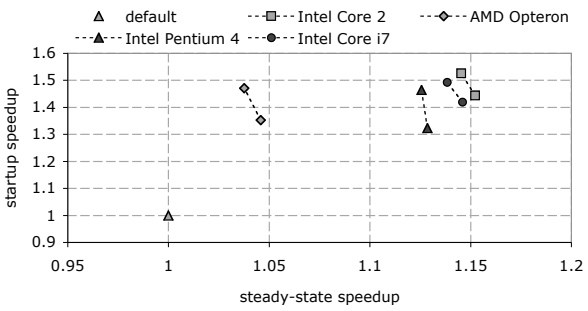


Figure 7: Graph (a) shows compilation rate versus performance speedup for the Pareto-optimal compilation plans determined on the AMD Opteron, Pentium 4 and Core i7 when run on the Core 2 platform. Graph (b) shows start-up versus steady-state performance for the AMD Opteron, Pentium 4 and Core i7 tuned JIT compilers when evaluated on the Core 2. These graphs consider `mtrt`.

the space, suggesting that a few optimizations might have a large effect.

Employing tuned JIT compilers across platforms. Using a JIT compiler that was tuned for a particular hardware platform on another hardware platform may yield suboptimal results. This is illustrated in Figure 7(a) where the Pareto-optimal plans tuned for `mtrt` for each platform have been evaluated on the Intel Core 2 platform. The optimization plans tuned for the Intel Pentium 4 and AMD Opteron platforms are suboptimal, i.e., they perform worse than the ones that were tuned for the Core 2. We observe a similar result when looking into tuned JIT compilers, see Figure 7(b) which compares the performance of a JIT compiler tuned for the Pentium 4, AMD Opteron and Core i7 when run on a Core 2 machine against a JIT compiler that was tuned on the Core 2. Clearly, the JIT compiler tuned for the Core 2 yields the best possible performance on the Core 2—the JIT compilers tuned for the other platforms perform worse. In particular, the JIT compiler that was tuned for the Core 2 yields approximately 5% better start-up performance and 10% better steady-state performance compared to the JIT compiler that was tuned for the Pentium 4. We thus conclude that *platform-specific JIT compiler tuning can yield substantial performance benefits and transferring JIT settings across platforms may lead to suboptimal performance*.

5.6 Exploration time

Finally, we discuss how much time is needed to complete the JIT compiler tuning.

Performing the first step for all of the DaCapo and SPECjvm98 benchmarks on the Core2 platform took 33 generations to converge. During each generation, 25 new optimization plans are constructed, each of which requires roughly 40 minutes to measure the compiler DNA. This means that about 550 machine hours are needed to run the first step of the tuning process to convergence. Note however, that this tuning process is embarrassingly parallel, i.e., all plans can be measured in parallel and independently from each other. Having sufficient machine resources available, this first step takes 22 hours only.

The second exploration step, which tunes the plan-to-level assignment and AOS, converges significantly faster: only 8 generations are required. Evaluating a single JIT compiler setting takes about 400 minutes on average. This results in an additional exploration time of about 1320 hours. Again, because each generation can be evaluated in parallel, the exploration can be performed in about 53 hours.

Thus, the entire exploration for a set of 16 benchmarks on a particular hardware platform takes around 75 hours or roughly 3 days. Performing the exploration for a single application only, as we did for the experiments described in Section 5.3, is a matter of hours.

6. Related Work

6.1 Dynamic optimization

Most modern Java virtual machines implement multiple levels of optimization, see for example [4, 6, 19, 24]. Since compilation time is an integral part of the total execution time in a dynamic compiler, it is of utmost importance to make a good trade-off between compilation time and code quality when proposing optimization levels in a optimizing dynamic compiler. Arnold et al. [6] and Ishizaki et al. [16] describe how optimization levels are determined manually for the Jikes RVM and IBM DB production VM, respectively.

Cavazos and O'Boyle [9] take a different approach to optimizing JIT compilers. They apply a different optimization plan for each method. The optimizations in these plans are determined by using a logistic regression function that predicts which optimizations are most useful for the given method based on bytecode features. They report speedups of 4%, 2% and 29% on average compared to optimizing all methods at the `-O0`, `-O1`, and `-O2` levels, respectively. When considering adaptive optimization, they report a 1% improvement over default Jikes RVM for SPECjvm98; for

the DaCapo benchmarks, they report a 4% average performance improvement. Our JIT tuning approach does not apply different compilation plans to individual methods which simplifies JIT compiler tuning. In addition, Cavazos and O'Boyle do not make the case that JIT compilers that are tuned for particular hardware platforms and/or applications can yield substantial performance benefits. In their follow-on work [10], Cavazos and O'Boyle use a genetic algorithm to automatically tune the heuristics of the inliner in a dynamic Java compiler. Our work is not limited to the inliner; we instead tune the entire JIT optimization system.

6.2 Multi-objective iterative compilation

The basic idea of iterative compilation is to explore the compiler optimization space by iteratively compiling and measuring the effectiveness of optimization sequences. A large body of work has been done on iterative compilation over the past few years, and many researchers have reported impressive results showing significant performance, energy or code size improvements over standard optimization sequences, see for example [1, 2, 8, 11, 17, 25].

What all of this prior work on iterative compilation has in common is that it focuses on a single objective function to be optimized. For example, researchers typically focus on a single optimization criterion such as performance [1, 8, 9, 11, 25], or energy consumption [14], or code size [11]. And some researchers focus on optimizing a single objective function that combines multiple optimization criteria such as code quality and compilation time [9, 10], or code quality and code size [17].

Very recently, Hoste and Eeckhout [15] proposed the COLE framework which explores a multi-objective compiler optimization space, unlike prior work which focuses on single-objective optimization. The key innovation compared to COLE is that this paper studies multi-objective compiler optimization in a dynamic compiler; the COLE work focused on static compilers for programming languages such as C, C++, Fortran, etc. A dynamic compiler poses several new challenges compared to a static compiler which complicates the search process, which, as mentioned before, motivated us for the two-step process proposed in this paper.

7. Conclusion

This paper proposed a framework for automatically tuning dynamic compilers. The framework uses evolutionary searching and tunes the JIT compiler for a given hardware platform and a given application or application domain. This is done through a two-step process in order to manage the complexity in exploring the huge optimization space: we first identify Pareto-optimal compilation plans, and subsequently assign plans to optimization levels and fine-tune the AOS. Our experimental results using the Jikes RVM, four hardware platforms and the SPECjvm98 and DaCapo benchmarks, demonstrate that the proposed framework identifies JIT compiler configurations that achieve significantly better performance compared to a manually tuned VM. When optimizing for individual applications, we achieve performance improvements up to 40% and 19% for start-up and steady-state performance, respectively. Also, optimizing for a specific hardware platform leads to significantly better performance. Our framework is completely automated and explores the complex JIT compiler space in approximately 3 days for the collection of DaCapo and SPECjvm benchmarks; tuning the JIT compiler for individual applications is done in a few hours.

Acknowledgements

We would like to thank Brad Chen and the anonymous reviewers for their thoughtful comments and valuable suggestions. Kenneth Hoste is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Andy Georges is supported through a postdoctoral fellowship by the Research Foundation–Flanders (FWO). Additional support is provided by the FWO projects G.0232.06, G.0255.08, and G.0179.10, and the UGent-BOF projects 01J14407 and 01Z04109. Computational resources and services used in this work were provided by Ghent University.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *CGO*, pages 295–305, Mar. 2006.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms. In *LCTES*, pages 231–239, June 2004.

[3] 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. In *IBM System Journal*, 39(1), Feb. 2000.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, Oct. 2000.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machine. In *Proceedings of the IEEE*, 93(2), 2005

[6] M. Arnold, M. Hind, and B. G. Ryder. Online Feedback-Directed Optimization in Java. In *OOPSLA*, pages 111–129, 2002

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, Oct. 2006.

[8] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *PACT*, Oct. 1998.

[9] J. Cavazos, and M. O'Boyle Method-Specific Dynamic Compilation using Logistic Regression. In *OOPSLA*, pages 229–240, Oct. 2006.

[10] J. Cavazos and M. O'Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*, Nov. 2005.

[11] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES*, pages 1–9, May 1999.

[12] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *OOPSLA*, pages 169–186, Oct. 2003.

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76, Oct. 2007.

[14] S. V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *Journal of Embedded Computing*, 1(4):509–520, July 2005.

[15] K. Hoste and L. Eeckhout. COLE: Compiler Optimization Level Exploration In *CGO*, pages 165–174, Apr. 2008.

[16] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, T. Nakatani Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *OOPSLA*, pages 187–204, Oct. 2003.

[17] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI*, pages 171–182, June 2004.

[18] Y. Luo, and L.K. John. Efficiently Evaluating Speedup Using Sampled Processor Simulation. In *IEEE Computer Architecture Letters*, 3, 2004

[19] D. Maier, P. Ramarao, M. Stoodley, and V. Sundaresan. Experiences with Multithreading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *CGO*, pages 87–97, Mar. 2006.

[20] J. Neter, M. H. Kutner, W. Wasserman, and C. J. Nachtsheim. *Applied Linear Statistical Models*. McGraw-Hill, 1996.

[21] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *JVM*, pages 1–12, Apr. 2001.

[22] Standard Performance Evaluation Corporation. SPECjbb2000 Benchmark. http://www.spec.org/jbb2000.

[23] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

[24] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a Java Just-In-Time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):732–785, July 2005.

[25] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. *Journal of Instruction-level Parallelism*, Jan. 2005. Accessible at `http://www.jilp.org/vol7`.

[26] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. In IEEE Transactions on Evolutionary Computation, 3(4), pages 257–271, Nov. 1999.

[27] E. Zitzler, M. Laumanns and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report TIK-Report 103, May 2001.