# Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation

Michael R. Jantz  Prasad A. Kulkarni

Electrical Engineering and Computer Science, University of Kansas

{mjantz,kulkarni}@ittc.ku.edu

## Abstract

*Phase selection* is the process of customizing the applied set of compiler optimization phases for individual functions or programs to improve performance of generated code. Researchers have recently developed novel feature-vector based heuristic techniques to perform phase selection during online JIT compilation. While these heuristics improve program *startup* speed, *steady-state* performance was not seen to benefit over the default fixed single sequence baseline. Unfortunately, *it is still not conclusively known whether this lack of steady-state performance gain is due to a failure of existing online phase selection heuristics, or because there is, indeed, little or no speedup to be gained by phase selection in online JIT environments*. The goal of this work is to resolve this question, while examining the phase selection related behavior of optimizations, and assessing and improving the effectiveness of existing heuristic solutions.

We conduct experiments to find and understand the potency of the factors that can cause the phase selection problem in JIT compilers. Next, using long-running genetic algorithms we determine that program-wide and method-specific phase selection in the HotSpot JIT compiler can produce *ideal* steady-state performance gains of up to 15% (4.3% average) and 44% (6.2% average) respectively. We also find that existing state-of-the-art heuristic solutions are unable to realize these performance gains (in our experimental setup), discuss possible causes, and show that exploiting knowledge of optimization phase behavior can help improve such heuristic solutions. Our work develops a robust open-source production-quality framework using the HotSpot JVM to further explore this problem in the future.

***Categories and Subject Descriptors*** D.3 [*Software*]: Programming languages; D.3.4 [*Programming languages*]: Processors—Compilers, Optimizations; I.2.6 [*Artificial intelligence*]: Learning—Induction

***General Terms*** Performance, Experimentation, Languages

***Keywords*** Phase selection, Compiler optimizations, HotSpot

## 1. Introduction

An optimization phase in a compiler transforms the input program into a semantically equivalent version with the goal of improving the performance of generated code. Quality compilers implement many optimizations. Researchers have found that the set of optimizations producing the best quality code varies for each method/program and can result in substantial performance benefits over always applying any single optimization sequence [5, 12]. *Optimization phase selection* is the process of automatically finding the best set of optimizations for each method/program to maximize performance of generated code, and is an important, fundamental, but unresolved problem in compiler optimization.

We distinguish phase selection from the related issue of *phase ordering*, which explores the effect of different orderings of optimization phases on program performance. We find that although phase ordering is possible in some research compilers, such as VPO [17] and Jikes Research Virtual Machine (VM) [19], reordering optimization phases is extremely hard to support in most production systems, including GCC [8] and the HotSpot VM, due to their use of multiple intermediate formats and complex inherent dependencies between optimizations. Therefore, our work for this paper conducted using the HotSpot JVM only explores the problem of phase selection by selectively turning optimization phases ON and OFF.

The problem of optimization selection has been extensively investigated by the static compiler community [2, 8, 12, 25]. Unfortunately, techniques employed to successfully address this problem in static compilers (such as *iterative compilation*) are not always applicable to dynamic or JIT (Just-in-Time) compilers. Since compilation occurs at runtime, one over-riding constraint for dynamic compilers is the need to be fast so as to minimize interference with program execution and to make the optimized code available for execution sooner. To realize fast online phase selection, researchers have developed novel techniques that employ feature-vector based machine-learning heuristics to quickly customize the set of optimizations applied to each method [5, 23]. Such heuristic solutions perform the time-consuming task of *model learning* offline, and then use the models to quickly customize optimization sets for individual programs/methods online during JIT compilation.

It has been observed that while such existing online phase selection techniques improve program *startup* (application + compilation) speed, they do not benefit throughput or *steady-state* performance (program speed after all compilation activity is complete). Program throughput is very important to many longer-running applications. Additionally, with the near-universal availability of increasingly parallel computing resources in modern multi/many-core processors and the ability of modern VMs to spawn multiple asynchronous compiler threads, researchers expect the compilation overhead to become an even smaller component of the total program runtime [16]. Consequently, achieving steady-state or *code-quality* improvements is becoming increasingly important for modern systems. Unfortunately, while researchers are still investing in the development of new techniques to resolve phase selection, *we do not yet conclusively know whether the lack of steady-state performance gain provided by existing online phase selection techniques is a failure of these techniques, or because there is, indeed, little or no speedup to be gained by phase selection in online environments*. While addressing this primary question, we make the following contributions in this work.

1. We conduct a thorough analysis to understand the phase selection related behavior of JIT compiler optimization phases and identify the potency of factors that could produce the phase selection problem,
2. We conduct long-running (genetic algorithm based) iterative searches to determine the *ideal* benefits of phase selection in online JIT environments,
3. We evaluate the accuracy and effectiveness of existing state-of-the-art online heuristic techniques in achieving the benefit delivered by (the more expensive) iterative search techniques and discuss improvements, and
4. We construct a robust open-source framework for dynamic JIT compiler phase selection exploration in the standard Sun/Oracle HotSpot Java virtual machine (JVM) [21]. Our framework prepares 28 optimizations in the HotSpot compiler for individual fine-grain control by different phase selection algorithms.

The rest of the paper is organized as follows. We present related work in the next section. We describe our HotSpot based experimental framework and benchmark set in Section 3. We explore the behavior of HotSpot compiler optimizations and present our observations in Section 4. We present our results on the ideal case performance benefits of phase sequence customization at the whole-program and per-method levels in Section 5. We determine the effectiveness of existing feature-vector based heuristic techniques and provide feedback on improving them in Section 6. Finally, we present our planned future work and the conclusions of this study in Sections 7 and 8 respectively.

## 2. Background and Related Work

In this section we provide some overview and describe related works in the area of optimization phase selection research. Phase selection and ordering are long-standing problems in compiler optimization. Several studies have also discovered that there is no single optimization set that can produce optimal code for every method/program [2]. A common technique to address the phase selection problem in static compilers is to iteratively evaluate the performance of many/all different phase sequences to find the best one for individual methods or programs. Unfortunately, with the large number of optimizations present in modern compilers (say, $n$), the search space of all possible combinations of optimization settings ($2^n$) can be very large. Therefore, researchers have in the past employed various techniques to reduce the space of potential candidate solutions. Chow and Wu applied a technique called fractional factorial design to systematically design a series of experiments to select a good set of program-specific optimization phases by determining interactions between phases [6]. Haneda et al. employed statistical analysis of the effect of compiler options to prune the phase selection search space and find a single compiler setting for a collection of programs that performs better than the standard settings used in GCC [11]. Pan and Eigenmann developed three heuristic algorithms to quickly select good compiler optimization settings, and found that their combined approach that first identifies phases with negative performance effects and greedily eliminates them achieves the best result [22]. Also related is the work by Fursin et al. who develop a GCC-based framework (MILEPOST GCC) to automatically extract program *features* and learn the best optimizations across programs. Given a new program to compile, this framework can correlate the program's features with the closest program seen earlier to apply a customized and potentially more effective optimization combination [8].

Machine-learning techniques, such as genetic algorithms and hill-climbing, have been commonly employed to search for the best set or ordering of optimizations for individual programs/methods. Cooper et al. were among the first to use machine-learning algorithms to quickly and effectively search the phase selection search space to find program-level optimization sequences to reduce code-size for embedded applications [2, 7]. Hoste and Eeckhout developed a system called *COLE* that uses genetic algorithm based multi-objective evolutionary search algorithm to automatically find pareto optimal optimization settings for GCC [12]. Kulkarni et al. compared the proficiency of several machine-learning algorithms to find the best phase sequence as obtained by their exhaustive search strategy [17]. They observed that search techniques such as genetic algorithms achieve benefit that is very close to that best performance. *We use this result in our current study to characterize the GA-delivered best performance as a good indicator of the performance limit of phase selection in dynamic compilers.*

Also related is their more recent work that compares the ability of GA-based program and function-level searches to find the best phase sequence, and finds the finer-granularity of function-level searches to achieve better overall results in their static C compiler, VPO [18]. All these above studies were conducted for static compilers. Instead, in this work, we use the GA searches to determine the performance limits of phase selection in dynamic compilers.

While program-specific GA and other iterative searches may be acceptable for static compilers, they are too time-consuming for use in dynamic compilers. Consequently, researchers have developed novel techniques to quickly customize phase sequences to individual methods in dynamic JIT compilers. Cavazos and O'Boyle employed the technique of logistic regression to learn a predictive model offline that can later be used in online compilation environments (like their Jikes Research VM) to derive customized optimization sequences for methods based on its features [5]. Another related work by Sanchez et al. used support vector machines to learn and discover method-specific compilation sequences in IBM's (closed-source) production JVM [23]. This work used a different learning algorithm and was conducted in a production JVM. While these techniques reduce program startup times due to savings in the compilation overhead, the feature-vector based online algorithm was not able to improve program steady-state performance over the default compiler configuration. Additionally, none of the existing works attempt to determine the potential benefit of phase selection to improve code-quality in a dynamic compiler. While resolving this important question, we also evaluate the success of existing heuristic schemes to achieve this best potential performance benefit in the HotSpot production JVM.

In this paper, we also investigate the behavior of optimizations to better understand the scope and extent of the optimization selection problem in dynamic compilers. Earlier researchers have attempted to measure the benefit of dynamic optimizations, for example, to overcome the overheads induced by the safety and flexibility constraints of Java [14], in cross-platform implementations [15], and to explore optimization synergies [20]. However, none of these works perform their studies in the context of understanding the optimization selection problem for JIT compilers.

## 3. Experimental Framework

In this section, we describe the compiler and benchmarks used, and the methodology employed for our experiments.

| Optimization Phase | Description |
|---|---|
| aggressive copy coalescing | Perform aggressive copy coalescing after coming out of SSA form (before register allocation). |
| block layout by frequency | Use edge frequencies to drive block ordering. |
| block layout rotate loops | Allow back branches to be fall through when determining block layout. |
| conditional constant prop. | Perform optimistic sparse conditional constant propagation until a fixed point is reached. |
| conservative copy coalescing | Perform conservative copy coalescing during register allocation (RA). Requires RA is enabled. |
| do escape analysis | Identify and optimize objects that are accessible only within one method or thread. |
| eliminate allocations | Use escape analysis to eliminate allocations. |
| global code motion | Hoist instructions to block with least execution frequency. |
| inline | Replace (non accessor/mutator) method calls with the body of the method. |
| inline accessors | Replace accessor/mutator method calls with the body of the method. |
| instruction scheduling | Perform instruction scheduling after register allocation. |
| iterative global value numbering (GVN) | Iteratively replaces nodes with their values if the value has been previously recorded (applied in several places after parsing). |
| loop peeling | Peel out the first iteration of a loop. |
| loop unswitching | Clone loops with an invariant test and insert a clone of the test that selects which version to execute. |
| optimize null checks | Detect implicit null check opportunities (e.g. null checks with suitable memory operations nearby use the memory operation to perform the null check). |
| parse-time GVN | Replaces nodes with their values if the value has been previously recorded during parsing. |
| partial loop peeling | Partially peel the top portion of a loop by cloning and placing one copy just before the new loop head and the other copy at the bottom of the new loop (also known as loop rotation). |
| peephole remove copies | Apply peephole copy removal immediately following register allocation. |
| range check elimination | Split loop iterations to eliminate range checks. |
| reassociate invariants | Re-associates expressions with loop invariants. |
| register allocation | Employ a Briggs-Chaitin style graph coloring register allocator to assign registers to live ranges. |
| remove useless nodes | Identify and remove useless nodes in the ideal graph after parsing. |
| split if blocks | Folds some branches by cloning compares and control flow through merge points. |
| use loop predicate | Generate a predicate to select fast/slow loop versions. |
| use super word | Transform scalar operations into packed (super word) operations. |
| eliminate auto box$^*$ | Eliminate extra nodes in the ideal graph due to autoboxing. |
| optimize fills$^*$ | Convert fill/copy loops into an intrinsic method. |
| optimize strings$^*$ | Optimize strings constructed by StringBuilder. |

**Table 1.** Configurable optimizations in our modified HotSpot compiler. Optimizations marked with $*$ are disabled in the default compiler.

## 3.1 Compiler and Benchmarks

We perform our study using the server compiler in Sun/Oracle's HotSpot Java virtual machine (build 1.6.0_25-b06) [21]. Similar to many production compilers, *HotSpot imposes a strict ordering on optimization phases* due to the use of multiple intermediate representations and documented or undocumented assumptions and dependencies between different phases. Additionally, the HotSpot compiler applies a fixed set of optimizations to every method it compiles. The compilation process parses the method's bytecodes into a static single assignment (SSA) representation known as the *ideal graph*. Several optimizations, including method inlining, are applied as the method's bytecodes are parsed. The compiler then performs a fixed set of optimizations on the resultant structure before converting to machine instructions, performing scheduling and register allocation, and finally generating machine code.

The HotSpot JVM provides command-line flags to optionally enable or disable several optimizations. However, many optimization phases do not have such flags, and some also generate/update analysis information used by later stages. We modified the HotSpot compiler to provide command-line flags for most optimization phases, and factored out the analysis calculation so that it is computed regardless of the optimization setting. Some transformations, such as *constant folding* and *instruction selection*, are required by later stages to produce correct code and are hard to effectively disable. We also do not include a flag for *dead code elimination*, which is performed continuously by the structure of the intermediate representation and would require much more invasive changes to disable. We perform innovative modifications to deactivate the compulsory phases of *register allocation* (RA) and *global code motion* (GCM). The disabled version of RA assumes every register conflicts with every live range, and thus, always spills live ranges to memory. Likewise, the disabled version of GCM schedules all instructions to execute as late as possible and does not attempt to hoist instructions to blocks that may be executed much less frequently. Finally, we modified the HotSpot JVM to accept binary sequences describing the application status (ON/OFF) of each phase at the program or method-level. Thus, *while not altering any optimization algorithm or the baseline compiler configuration, we made several major updates to the HotSpot JIT compiler to facilitate its use during phase selection research.* Table 1 shows the complete set of optimization phases that we are now able to optionally enable or disable for our experiments.

Our experiments were conducted over applications from two suites of benchmarks. We use all SPECjvm98 benchmarks [24] with two input sizes (10 and 100), and 12 (of 14) applications from the DaCapo benchmark suite [4] with the small and default inputs. Two DaCapo benchmarks, *tradebeans* and *tradesoap*, are excluded from our study since they do not always run correctly with our *default* HotSpot VM.

## 3.2 Performance Measurement

One of the goals of this research is to quantify the performance benefit of optimization phase selection *with regards to generated code quality* in a production-grade dynamic compiler. Therefore, the experiments in this study discount compilation time and measure the *steady-state* performance. In the default mode, the VM employs *selective compilation* and only compiles methods with execution counts that exceed the selected threshold. Our experimental setup first determines this set of (hot) methods compiled during the *startup* mode for each benchmark. All our steady-state experiments only compile this hot method set for all its program runs. Both the SPECjvm98 and DaCapo harness allow each benchmark to be iterated multiple times in the same VM run. During our steady-state program runs we disable background compilation to force all these hot methods to be compiled in the first program iteration. We modify our VM to reset execution counts after each iteration to prevent methods from becoming hot (and getting compiled) in later iterations. We allow each benchmark to iterate five more times and record the median runtime of these iterations as the steady-state program run-time. To account for inherent timing variations during the benchmark runs, *all the performance results in this paper report the average and 95% confidence intervals over* ten *steady-state runs* using the setup described by Georges et al. [9].

All experiments were performed on a cluster of Dell PowerEdge 1850 server machines running Red Hat Enterprise Linux 5.1 as the operating system. Each machine has four 64-bit 2.8GHz Intel Xeon processors, 6GB of DDR2 SDRAM, and a 4MB L2 cache. Our HotSpot VM uses the stand-alone server compiler and the default garbage collector settings for "server-class" machines [13] ("parallel collector" GC, initial heap size is 96MB, maximum is 1GB). We make no attempt to restrict or control GC during our experiments. Finally, there are no hyperthreading or frequency scaling techniques of any kind enabled during our experiments.

## 4. Analyzing Behavior of Compiler Optimizations for Phase Selection

Compiler optimizations are designed to improve program performance. Therefore, it is often (naïvely) expected that always applying (turning ON) all available optimizations to all program regions should generate the best quality code. However, optimizations operating on the same program code and competing for finite machine resources (registers) may interact with each other. Such interactions may remove opportunities for later optimizations to generate even better code. Additionally, program performance is often very hard for the compiler to predict on the current generation of machines with complex architectural and micro-architectural features. Consequently, program transformations performed by an optimization may not always benefit program execution speed. The goal of effective phase selection is to find and disable optimizations with negative effects for each program region. In this section we conduct a series of experiments to explore important optimization selection issues, such as why and when is optimization selection effective for standard dynamic JIT compilers. We are also interested in finding indicators to suggest that customizing optimization selections for individual programs or methods is likely to benefit performance. We report several interesting observations that help explain both the prior as well as our current results in phase selection research for dynamic JIT compilers.

### 4.1 Experimental Setup

Our setup to analyze the behavior of optimization phases is inspired by Lee et al.'s framework to determine the benefits and costs of compiler optimizations [20]. Our experimental configuration (*defOpt*) uses the default HotSpot server compilation sequence as baseline. The execution time of each benchmark with this baseline ($T(OPT < defOpt >)$) is compared with its time obtained by a JIT compiler that disables one optimization ($x$) at a time ($T(OPT < defOpt - x >)$). We use the following fraction to quantify the effect of HotSpot optimizations in this configuration.

$$\frac{T(OPT < defOpt - x >) - T(OPT < defOpt >)}{T(OPT < defOpt >)} \qquad (1)$$

Each experimental run disables only one optimization (out of 25) from the optimization set used in the default HotSpot compiler. Equation 1 computes a negative value if removing the corresponding optimization, $x$, from the baseline optimization set improves performance (reduces program runtime) of the generated code. In other words, a negative value implies that including that optimization harms performance of the generated code. The HotSpot JIT compiler uses an individual method for its compilation unit. Therefore, in this section we evaluate the effect of compiler optimizations over distinct program methods.

Our experiments in this section are conducted over 53 *hot* focus methods over all the programs in our benchmark suite. These focus methods are selected because each comprises *at least* 10% of its respective benchmark run-time. More details on the rationale and selection of focus methods, as well as a complete list of these methods, are provided in Section 5.3.1.

### 4.2 Results and Observations

Figure 1 (left Y-axis, bar-plot) illustrates the *accumulated* negative and positive impact of each optimization calculated using Equation 1 over all our 53 individual program methods. For each HotSpot optimization, Figure 1 (right Y-axis, line-plot) also shows the number of program methods that witness a negative or positive impact. These results enable us to make several important observations regarding the behavior of optimizations in the HotSpot JIT compiler. **First**, the results validate the claims that optimizations are not always beneficial to program performance. This observation provides the motivation and justification for further developing and exploring effective phase selection algorithms to enable the JIT compiler to generate the best possible output code for each method/program. **Second**, we observe that *most* optimizations in the HotSpot JIT compiler produce, at least occasional, negative effects. This observation indicates that eliminating the optimization phase selection issue may require researchers to understand and update several different compiler optimizations, which makes a compiler design-time solution very hard. **Third**, most optimizations do not negatively impact a large number of program methods, and the typical negative impact is also not very high. However, we also find optimizations, including *AggressiveCoalesce*, *IterGVN*, and *SplitIfBlocks*, that, rather than improving, show a degrading performance impact more often. This result is surprising since dynamic compilers generally only provide the more conservative compiler optimizations.[1] Thus, this study finds optimizations that need to be urgently analyzed to alleviate the optimization selection problem in HotSpot. **Fourth**, we unexpectedly find that most of the optimizations in the HotSpot JIT compiler only have a marginal individual influence on performance. We observe that *method inlining* is by far the most beneficial compiler optimization in HotSpot, followed by *register allocation*.[2]

Figure 2 plots the accumulated positive and negative optimization impact (on left Y-axis, bar-plot) and the number of optimizations that impact performance (on right Y-axis, line-plot) for each of our focus methods represented along the X-axis. These results allow us to make two other observations that are particularly enlightening. **First**, there are typically not many optimizations that

[1] Compare the 28 optimization flags in HotSpot with over 100 such flags provided by GCC.

[2] Method inlining is a difficult optimization to control. Our experimental setup, which uses a fixed list of methods to compile, may slightly exaggerate the performance impact of disabling method inlining because some methods that would normally be inlined may not be compiled at all if they are not in the hot method list. To avoid such exaggeration, one possibility is to detect and compile such methods when inlining is disabled. However, an inlined method (say, P) that is not otherwise compiled spends its X inlined invocations in compiled code, but other Y invocations in interpreted code. With inlining disabled for the focus method, if P is compiled then it will spend all 'X+Y' invocations in compiled code. We chose the exaggeration because we found that it was very uncommon for methods not in the fixed list to still be inlinable.

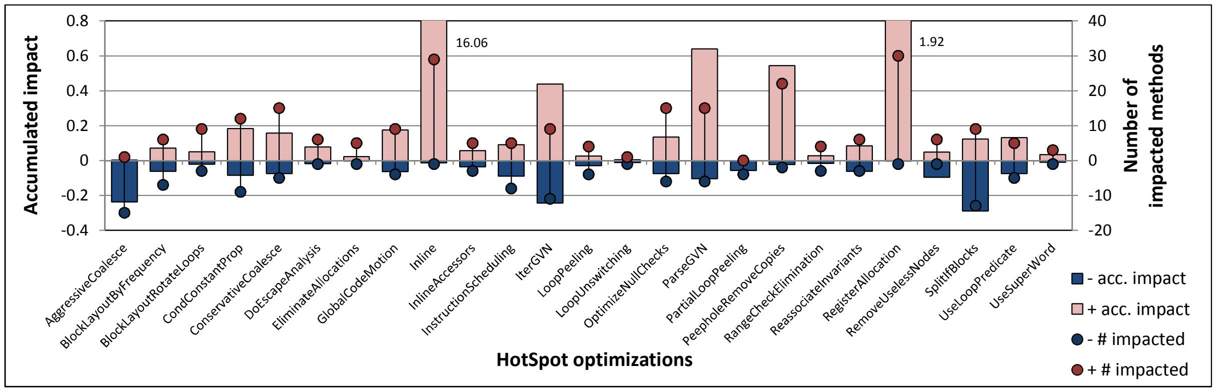


**Figure 1. Left Y-axis:** Accumulated positive and negative impact of each HotSpot optimization over our focus methods (non-scaled). **Right Y-axis:** Number of focus methods that are positively or negatively impacted by each HotSpot optimization.

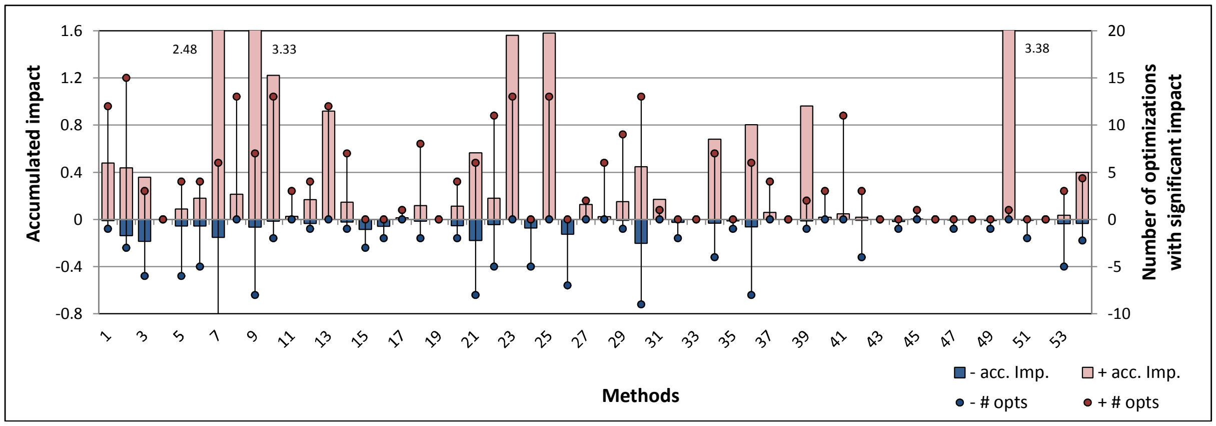


**Figure 2. Left Y-axis:** Accumulated positive and negative impact of the 25 HotSpot optimizations for each focus method (non-scaled). **Right Y-axis:** Number of optimizations that positively or negatively impact each focus method.

degrade performance for any single method (2.2 out of 25, on average). More importantly, even for methods with several individual degrading optimizations, the accumulated negative impact is never very high. This result, in a way, tempers the expectations of performance improvements from ideal phase selection in JIT compilers (particularly, HotSpot). In other words, we can only expect customized phase selections to provide modest performance benefits for individual methods/programs in most cases. **Second**, for most methods, there are only a few optimizations (4.36 out of 25, on average) that benefit performance. Thus, there is a huge potential for saving compilation overhead during program *startup* by disabling the *inactive* optimizations. It is this, hitherto unreported, attribute of JIT compilers that enables the online feature-vector based phase selection algorithms to improve program startup performance in earlier works [5, 23].

Finally, we note that although this simple study provides useful information regarding optimization behavior, it may not capture all possible optimization interactions that can be simultaneously active in a single optimization setting for a method. For example, phase interactions may cause compiler optimization phases that degrade performance when applied alone to improve performance when combined with other optimizations. However, these simple experiments provided us with both the motivation to further explore the potential of phase selection for dynamic compilers, while lowering our expectations for large performance benefits.

## 5. Limits of Optimization Selection

Most dynamic JIT compilers apply the same set of optimization phases to all methods and programs. Our results in the last section indicate the potential for performance gains by customizing optimization phase selection for individual (smaller) code regions. In this section we conduct experiments to quantify the steady-state speed benefits of customizing optimization sets for individual programs/methods in JIT compilers. The large number of optimizations in HotSpot makes it unfeasible to perform *exhaustive* optimization selection search space evaluation. Earlier research has demonstrated that genetic algorithms (GA) are highly effective at finding near-optimal phase sequences [17]. Therefore, we use a variant of a popular GA to find effective program-level and method-level optimization phase selection solutions [7]. Correspondingly, *we term the benefit in program run-time achieved by the GA derived phase sequence over the default HotSpot VM as the ideal performance benefit of phase selection for each program/method.*

We also emphasize that it is impractical to employ a GA-based solution to customize optimization sets in an online JIT compilation environment. Our program-wide and method-specific GA experiments are intended to only determine the performance limits of phase selection. We use these limits in the next section to evaluate the effectiveness of existing state-of-the-art heuristics to specialize optimization sets in online JIT compilers.

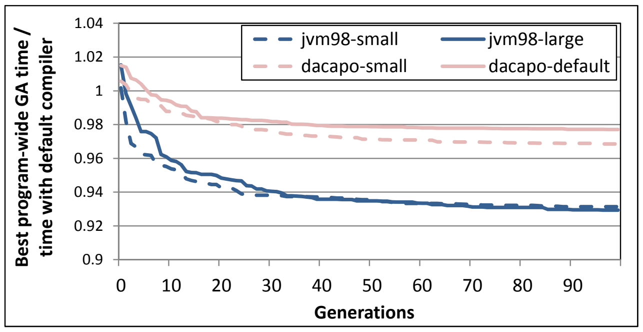


**Figure 3.** Average performance of best GA sequence in each generation compared to the default compiler.

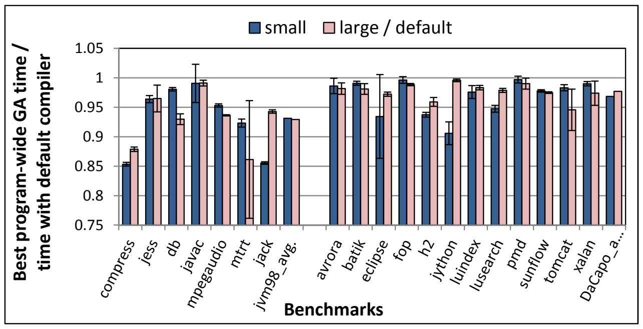


**Figure 4.** Performance of best program-wide optimization phase sequence after 100 generations of genetic algorithm.

### 5.1 Genetic Algorithm Description

In this section we describe the genetic algorithm we employ for our phase selection experiments. Genetic algorithms are heuristic search techniques that mimic the process of natural evolution [10]. *Genes* in the GA correspond to binary digits indicating the ON/OFF status of an optimization. *Chromosomes* correspond to optimization phase selections. The set of chromosomes currently under consideration constitutes a *population*. The evolution of a population occurs in *generations*. Each generation evaluates the *fitness* of every chromosome in the current population, and then uses the operations of *crossover* and *mutation* to create the next population. The number of *generations* specifies the number of population sets to evaluate. Chromosomes in the first GA generation are randomly initialized. After evaluating the performance of code generated by each chromosome in the *population*, they are sorted in decreasing order of performance. During *crossover*, 20% of chromosomes from the poorly performing half of the population are replaced by repeatedly selecting two chromosomes from the better half of the population and replacing the lower half of each chromosome with the lower half of the other to produce two new chromosomes each time. During *mutation* we flip the ON/OFF status of each gene with a small probability of 5% for chromosomes in the upper half of the population and 10% for the chromosomes in the lower half. The chromosomes replaced during crossover, as well as (up to five) chromosome(s) with performance(s) within one standard deviation of the best performance in the generation are not mutated. The *fitness criteria* used by our GA is the steady-state performance of the benchmark. For this study, we have 20 chromosomes in each population and run the GA for 100 generations. We have verified that 100 generations are sufficient for the GA to reach saturation in most cases. To speed-up the GA runs, we developed a parallel GA implementation that can simultaneously evaluate multiple chromosomes in a generation on a cluster of identically-configured machines.

### 5.2 Program-Wide GA Results

In this experiment we use our GA to find unique optimization selections for each of our benchmark-input pairs. Figure 3 plots the performance of code compiled with the best optimization set found by the GA in each generation as compared to the code generated by the default compiler sequence averaged over all programs for each benchmark suite. This figure shows that most (over 75%) of the average performance gains are realized in the first few (20) GA generations. Also, over 90% of the best average performance is obtained after 50 generations. Thus, 100 generations seem sufficient for our GA to converge on its near-best solution for most benchmarks. We also find that the SPECjvm98 benchmarks benefit more from optimization specialization than the DaCapo benchmarks. As expected, different inputs do not seem to significantly affect the steady-state optimization selection gains for most benchmarks.

Figure 4 compares the performance of each program optimized with the best program-wide optimization phase set found by our genetic algorithm with the program performance achieved by the default HotSpot server compiler for both our benchmark suites. The error bars show the 95% confidence interval for the difference between the means over 10 runs of the best customized optimization selection and the default compiler sequence. We note that the default optimization sequence in the HotSpot compiler has been heavily tuned over several years to meet market expectations for Java performance, and thus presents a very aggressive baseline. In spite of this aggressively tuned baseline, we find that customizing optimization selections can significantly improve performance (up to 15%) for several of our benchmark programs.

On average, the SPECjvm98 benchmarks improve by about 7% with both their *small* and *large* inputs. However, for programs in the DaCapo benchmark suite, program-wide optimization set specialization achieves smaller average benefits of 3.2% and 2.3% for their *small* and *default* input sizes respectively. The DaCapo benchmarks typically contain many more *hot* and *total* program methods as compared to the SPECjvm98 benchmarks. Additionally, unlike several SPECjvm98 programs that have a single or few dominant hot methods, most DaCapo benchmarks have a relatively flat execution profile with many methods that are similarly hot, with only slightly varying degrees [4]. Therefore, program-wide optimization sets for DaCapo benchmarks are customized over much longer code regions (single optimization set over many more hot methods), which, we believe, results in lower average performance gains from program-wide optimization selection. Over all benchmarks, the average benefit of ideal program-wide phase selection is 4.3%.

### 5.3 Method-Specific Genetic Algorithm

The default HotSpot compiler optimizes individual methods at a time, and applies the same set of optimizations to each compiled method. Prior research has found that optimization phase sequences tuned to each method yield better program performance than a single program-wide phase sequence [1, 18]. In this section, we explore the performance potential of optimization selection at the method-level during dynamic JIT compilation.

#### 5.3.1 Experimental Setup

There are two possible approaches for implementing GA searches to determine the performance potential of method-specific optimization phase settings: (a) running multiple simultaneous (and independent) GAs to gather optimization sequences for all program methods *concurrently* in the same run, and (b) executing the GA for each method *in isolation* (one method per program run). The first

| # | Benchmark | Method | % Time |
|---|---|---|---|
| 1 | db-large | Database.shell_sort | 86.67 |
| 2 | compress-small | Compressor.compress | 54.99 |
| 3 | compress-large | Compressor.compress | 53.42 |
| 4 | avrora-default | LegacyInterpreter.fastLoop | 50.85 |
| 5 | db-small | Database.shell_sort | 50.72 |
| 6 | jess-small | Node2.findInMemory | 48.57 |
| 7 | jack-small | TokenEngine.getNextTokenFromStream | 48.05 |
| 8 | avrora-small | LegacyInterpreter.fastLoop | 44.49 |
| 9 | jack-large | TokenEngine.getNextTokenFromStream | 44.23 |
| 10 | sunflow-default | KDTree.intersect | 40.52 |
| 11 | luindex-default | DocInverterPerField.processFields | 40.43 |
| 12 | sunflow-default | TriangleMesh$WaldTriangle.intersect | 39.20 |
| 13 | sunflow-small | KDTree.intersect | 37.92 |
| 14 | sunflow-small | TriangleMesh$WaldTriangle.intersect | 36.78 |
| 15 | jess-large | Node2.runTestsVaryRight | 34.31 |
| 16 | jython-small | PyFrame.getlocal | 32.73 |
| 17 | luindex-small | DocInverterPerField.processFields | 30.51 |
| 18 | lusearch-small | SegmentTermEnum.scanTo | 29.88 |
| 19 | lusearch-default | SegmentTermEnum.scanTo | 28.76 |
| 20 | jess-large | Node2.runTests | 27.41 |
| 21 | compress-large | Decompressor.decompress | 24.86 |
| 22 | compress-small | Compressor.output | 23.39 |
| 23 | mpegaudio-small | q.l | 23.12 |
| 24 | batik-default | MorphologyOp.isBetter | 22.26 |
| 25 | mpegaudio-large | q.l | 21.87 |
| 26 | jython-small | PyFrame.setline | 21.79 |
| 27 | xalan-small | ToStream.characters | 21.70 |
| 28 | db-small | ValidityCheckOutputStream.strip1 | 21.52 |
| 29 | compress-large | Compressor.output | 21.40 |
| 30 | compress-small | Decompressor.decompress | 21.23 |
| 31 | xalan-default | ToStream.characters | 20.00 |
| 32 | pmd-default | DacapoClassLoader.loadClass | 19.26 |
| 33 | batik-small | PNGImageEncoder.clamp | 17.74 |
| 34 | sunflow-small | BoundingIntervalHierarchy.intersect | 15.22 |
| 35 | h2-small | Query.query | 13.84 |
| 36 | sunflow-default | BoundingIntervalHierarchy.intersect | 13.79 |
| 37 | javac-large | ScannerInputStream.read | 13.46 |
| 38 | javac-small | ScannerInputStream.read | 13.17 |
| 39 | luindex-small | TermsHashPerField.add | 13.01 |
| 40 | mpegaudio-small | tb.u0114 | 12.88 |
| 41 | jython-default | PyFrame.setline | 12.68 |
| 42 | mpegaudio-large | tb.u0114 | 12.61 |
| 43 | jess-large | Funcall.Execute | 12.25 |
| 44 | luindex-small | StandardTokenizerImpl.getNextToken | 12.23 |
| 45 | lusearch-small | IndexInput.readVLong | 11.82 |
| 46 | lusearch-default | StandardAnalyzer.tokenStream | 11.49 |
| 47 | lusearch-default | IndexInput.readVLong | 11.46 |
| 48 | lusearch-small | StandardAnalyzer.tokenStream | 11.44 |
| 49 | h2-default | Command.executeQueryLocal | 11.37 |
| 50 | luindex-default | TermsHashPerField.add | 10.65 |
| 51 | jython-default | PyFrame.getlocal | 10.62 |
| 52 | eclipse-default | Parser.parse | 10.52 |
| 53 | luindex-default | StandardTokenizerImpl.getNextToken | 10.49 |

**Table 2.** Focus methods and the % of runtime each comprises of their respective benchmark runs

approach requires instrumenting every program method to record the time spent in each method in a single program run. These individual method times can then be used to concurrently drive independent method-specific GAs for all methods in a program. The VM also needs the ability to use distinct optimization selections to be employed for different program methods. We implemented this experimental scheme for our HotSpot VM by updating the compiler to instrument each method with instructions that employ the x86 TSC (Time-Stamp Counter) to record individual method run-times. However, achieving accurate results with this scheme faces several challenges. The HotSpot JVM contains interprocedural optimizations, such as *method inlining*, due to which varying the optimization sequence of one method affects the performance behavior of other program methods. Additionally, we also found that the order in which methods are compiled can vary from one run of the program to the next, which affects optimization decisions and method run-times. Finally, the added instrumentation to record method times also adds some noise and impacts optimization application and method performance.

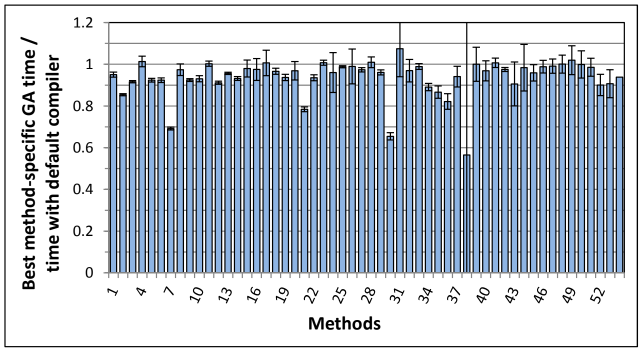
Therefore, we decided to employ the more straight-forward and accurate, but also time-consuming, approach of applying the GA to only one program method at a time. In each program run, the VM uses the optimization set provided by the GA to optimize one *focus method* and the default baseline set of optimizations to compile the other hot program methods. Thus, any reduction in the final program run-time over the baseline program performance can be attributed to the improvement in the single focus method. In an earlier *offline* run, we use our TSC based instrumentations with the baseline compiler configuration to estimate the fraction of total time spent by the program in each focus method. Any improvement in the overall program run-time during the GA is scaled with the fraction of time spent in the focus method to determine the run-time improvement in that individual method. We conduct this experiment over the 53 *focus methods* over all benchmarks that each comprise at least 10% of the time spent in their respective default program run. These methods, along with the % of total runtime each comprises in their respective benchmarks, are listed in Table 2.

### 5.3.2 Method-Specific GA Results

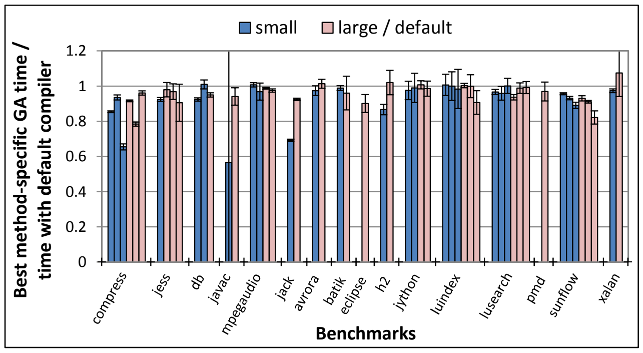
Figure 5(a) shows the *scaled* benefit in the run-time of each focus method when compiled with the best optimization set returned by the GA as compared to the method time if compiled with the baseline HotSpot server compiler. Methods along the $x$-axis in this graph are ordered by the fraction that they contribute to their respective overall program run-times (the same order methods are listed in Table 2). The final bar shows the average improvement over the 53 focus methods. Thus, we can see that customizing the optimization set for individual program methods can achieve significant performance benefits in some cases. While the best performance improvement is about 44%, method-specific optimization selection achieves close to a 6.2% reduction in run-time, on average. Figure 5(b) provides a different view of these same results with methods on the $x$-axis grouped together according to their respective benchmarks.

The plot in Figure 6 verifies that the (*non-scaled*) improvements in individual method run-times add-up over the entire program in most cases. That is, if individually customizing two methods in a program improves the overall program run-time by $x$ and $y$ respectively, then does the program achieve an $(x + y)$ percent improvement if both customized methods are used in the same program run? As mentioned earlier, our focus methods are selected such that each constitutes at least 10% of the baseline program run-time. Thus, different benchmarks contribute different number (zero, one, or more than one) of focus methods to our set. The first bar in Figure 6 simply sums-up the individual method run-time benefits (from distinct program runs) for *benchmarks that provide two or more focus methods*. The second bar plots the run-time of code generated using the best customized optimization sets for all focus methods *in the same run*. We print the number of focus methods provided by each benchmark above each set of bars. Thus, we find that the individual method benefits add-up well in many cases, yielding performance benefit that is close to the sum of the individual benefit of all its customized component methods.

Please note that the experiments for Figure 6 only employ customized optimization selections for the focus methods. The remaining hot benchmark methods are compiled using the baseline sequence, which results in lower average improvements as compared to the average in Figure 5(a). Thus, customizing optimization sets over smaller program regions (methods (6.2%) vs. programs (4.3%)) realize better overall performance gains for JIT compilers.



(a)



(b)

**Figure 5.** Performance of method-specific optimization selection after 100 GA generations. Methods in (a) are ordered by the % of run-time spent in their respective benchmarks. In (b), methods from the same benchmark are shown together. All results are are scaled by the fraction of total program time spent in the focus method and show the run-time improvement of that individual method.

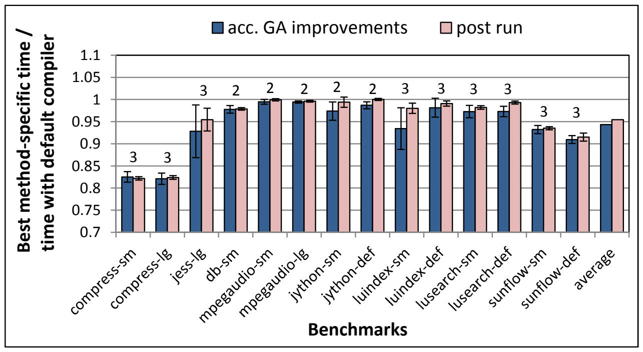


**Figure 6.** Accumulated improvement of method-specific optimization selection in benchmarks with multiple focus method.

It is important to note that we observe good correlation between the ideal method-specific improvements in Figure 5(a) and the per-method accumulated positive and negative impact of optimizations plotted in Figure 2. Thus, many methods with large accumulated negative effects (such as methods numbers #2, #3, #7, #21, #30, and #36) also show the greatest benefit from customized phase sequences found by our iterative (GA) search algorithm. Similarly, methods with small negative impacts in Figure 2 (including many methods numbered between #40 – #53) do not show significant benefits with ideal phase selection customization. While this correlation is encouraging, it may also imply that optimization interactions may not be very prominent in production-grade JVMs, such as HotSpot.

## 6. Effectiveness of Feature Vector Based Heuristic Techniques

Experiments in Sections 5.2 and 5.3 determine the potential gains due to effective phase selection in the HotSpot compiler. However, such iterative searches are extremely time-consuming, and are therefore not practical for dynamic compilers. Previous works have proposed using feature-vector based heuristic techniques to quickly derive customized optimization selections during online compilation [5, 23]. Such techniques use an expensive *offline* approach to construct their predictive models that are then employed by a fast *online* scheme to customize phase selections to individual methods. In this section we report results of the first evaluation (compared to *ideal*) of the effectiveness of such feature-vector based heuristic techniques for finding good optimization solutions.

### 6.1 Overview of Approach

Feature-vector based heuristic algorithms operate in two stages, *training* and *deployment*. The training stage conducts a set of offline experiments that measure the program performance achieved by different phase selections for a certain set of programs. This stage then selects the best performing sets of phases for each method. The approach then uses a set of program *features* to characterize every compilation unit (method). The features should be selected such that they are representative of the program properties that are important and relevant to the optimization phases, and are easy and fast to extract at run-time. Finally, the training stage employs statistical techniques, such as logistic regression and support vector machines, to correlate good optimization phase settings with the method feature list.

The deployment stage installs the learned statistical model into the compiler. Now, for each new compilation, the algorithm first determines the method's feature set. This feature set is given to the model that returns a customized setting for each optimization that is expected to be effective for the method. Thus, with this technique, each method may be compiled with a different phase sequence.

### 6.2 Our Experimental Configuration

We use techniques that have been successfully employed in prior works to develop our experimental framework [5, 23]. Table 3 lists the features we use to characterize each method, which are a combination of the features employed in earlier works and those relevant to the HotSpot compiler. These features are organized into two sets: *scalar features* consist of counters and binary attributes for a given method without any special relationship; *distribution features* characterize the actual code of the method by aggregating similar operand types and operations that appear in the method. The *counters* count the number of bytecodes, arguments, and temporaries present in the method, as well as the number of nodes in the intermediate representation immediately after parsing. *Attributes* include properties denoted by keywords (*final*, *protected*, *static*, etc.), as well as implicit properties such as whether the method contains loops or uses exception handlers. We record distribution features by incrementing a counter for each feature during bytecode parsing. The *types* features include Java native types, addresses (i.e. arrays) and user-defined objects. The remaining features correspond to one or more Java bytecode instructions. We use these features during the technique of logistic regression [3] to learn our model for

| Scalar Features | Distribution Features | | | |
|---|---|---|---|---|
| **Counters** | **Types** | | **ALU Operations** | |
| Bytecodes | byte | char | add | sub |
| Arguments | int | double | mul | div |
| Temporaries | short | long | rem | neg |
| Nodes | float | object | shift | or |
| | address | | and | xor |
| | | | inc | compare |
| **Attributes** | **Casting** | | **Memory Operations** | |
| Constructor | to byte | | load | load const |
| Final | to char | | store | new |
| Protected | to short | | new array / multiarray | |
| Public | to int | | | |
| Static | to long | | **Control Flow** | |
| Synchronized | to float | | branch | call |
| Exceptions | to double | | jsr | switch |
| Loops | to address | | | |
| | to object | | **Miscellaneous** | |
| | cast check | | instance of | throw |
| | | | array ops | field ops |
| | | | synchronization | |

**Table 3.** List of method features used in our experiments

these experiments. Logistic regression has the property that it can even output phase sequences not seen during the model-training. We have tuned our logistic regression model to make it as similar as possible to the one used previously by Cavazos and O'Boyle [5].

## 6.3 Feature-Vector Based Heuristic Algorithm Results

We perform two sets of experiments to evaluate the effectiveness of a feature-vector based logistic regression algorithm to learn and find good phase sequences for unseen methods during dynamic JIT compilation. As done in our other experiments, all numbers report the steady-state benchmark times. *All our experiments in this section employ cross-validation.* In other words, the evaluated benchmark or method (with both the small and large/default inputs) is never included in the training set for that benchmark or method.

Figure 7 plots the performance achieved by the optimization set delivered by the logistic regression algorithm when applied to each benchmark method as compared to the performance of the best benchmark-wide optimization sequence from Section 5.2. The training data for each program uses the top ten methods (based on their baseline run-times) from all the *other* (SPEC and DaCapo) benchmarks. While distinct benchmarks may contain different number of methods, we always consider ten methods from each program to weigh the benchmarks equally in the training set. For every benchmark, each top ten method contributes a distinct feature vector but uses the single benchmark-wide best optimization sequence from Section 5.2. The logistic regression algorithm may find a different optimization selection for each method during its online application. In spite of this flexibility, the feature vector based technique is never able to reach or improve the ideal single benchmark-wide optimization solution provided by the GA. Thus, figure 7 shows that, on average, the feature-vector based solution produces code that is 7.8% and 5.0% worse for SPECjvm98 (small and large data sets respectively) and 4.3% and 2.4% worse for DaCapo (small and default) as compared to the ideal GA phase selection. However, this technique is some times able to find optimization sequences that achieve performances that are close to or better than those realized by the default HotSpot server compiler. On average, the feature-vector based heuristic achieves performance that is 2.5% better for SPECjvm98-small benchmarks, and equal in all other cases (SPECjvm98-large and DaCapo small and default) as compared to the *baseline server compiler*.

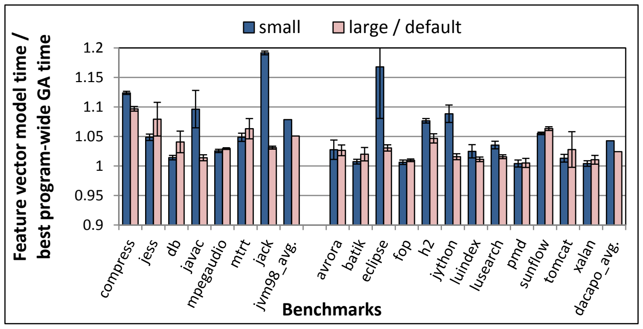


**Figure 7.** Effectiveness of benchmark-wide logistic regression. Training data for each benchmark consists of all the remaining programs from both benchmark suites.

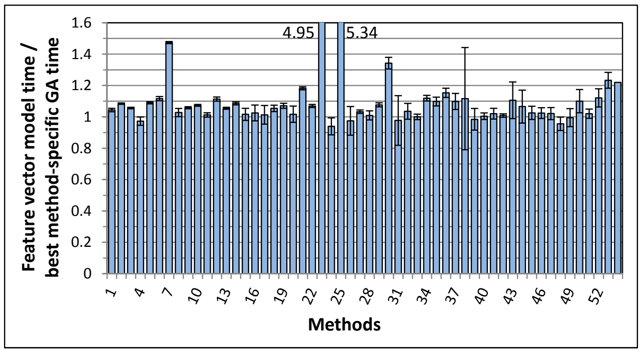
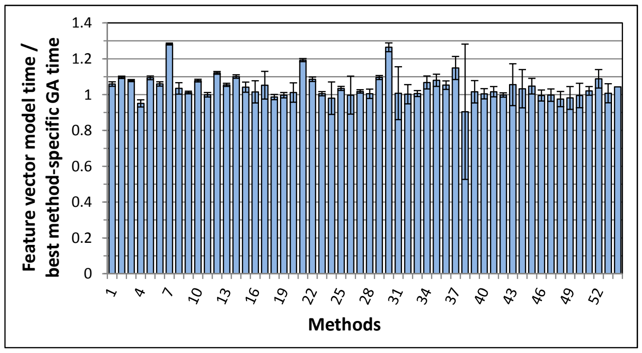


**Figure 8.** Effectiveness of method-specific logistic regression. Training data for each method consists of all the other focus methods used in Section 5.3.

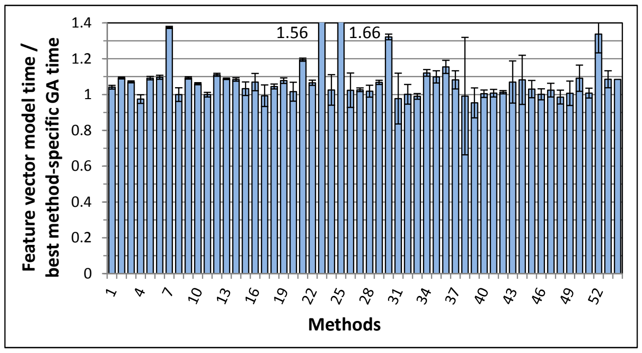
Figure 8 compares the performance of the logistic regression technique for individual program methods to their best GA-based performance. Since we employ cross-validation, the training data for each method uses information from all the other focus methods. Similar to the experimental configuration used in Section 5.3, each program run uses the logistic regression technique only for one *focus* method, while the remaining program methods are compiled with the baseline optimization sequence. The difference in program performance between the feature-vector based heuristic and the focus-method GA is scaled with the fraction of overall program time spent in the relevant method. Thus, we can see from Figure 8 that the per-method performance results achieved by the feature-vector based heuristic are quite disappointing. We find that, on average, the heuristic solutions achieve performance that is over 22% worse than the GA-tuned solution, and 14.7% worse than the baseline HotSpot server compiler.

## 6.4 Discussion

Thus, we find that existing state-of-the-art online feature-vector based algorithms are not able to find optimization sequences that improve code quality over the default baseline. We note that this observation is similar to the findings in other previous works [5, 23]. However, these earlier works did not investigate whether this lack of performance gain is because optimization selection is not especially beneficial in online JIT compilers, or if existing feature-vector heuristics are not powerful enough to realize those gains. Our experiments conclusively reveal that, although modest on average, the benefits of optimization customization do exist for several methods in dynamic compilers. Thus, additional research in



(a)



(b)

**Figure 9.** Experiments to analyze and improve the performance of feature-vector based heuristic algorithms for online phase selection. (a) Not using cross-validation and (b) Using observations for Section 4.

improving online phase selection heuristic algorithms is necessary to enable them to effectively specialize optimization settings for individual programs or methods. We conduct a few other experiments to analyze (and possibly, improve) the effectiveness of the logistic regression based feature-vector algorithm employed in this section.

In our first experiment we use the same per-method feature-vector based heuristic from the last section. However, instead of performing cross-validation, we allow the training data for each method to include that same method as well. Figure 9(a) plots the result of this experiment and compares the run-time of each method optimized with the phase selection delivered by the heuristic algorithm to the method's run-time when tuned using the ideal GA sequence. Thus, without cross-validation the heuristic algorithm achieves performance that is only 4.3% worse than ideal, and 2.5% *better* compared to the default HotSpot baseline. This result indicates that the logistic regression heuristic is not intrinsically poor, but may need a larger training set of methods, more subsets of methods in the training set with similar features that also have similar ideal phase sequences, and/or a better selection of method *features* to be more effective.

We have analyzed and observed several important properties of optimization phases in Section 4. In our next experiment, we employ the observation that most optimizations do not negatively impact a large number of methods to improve the performance of the feature-vector based heuristic (using cross-validation). With our new experiment we update the set of configurable optimizations (that we can set to ON/OFF) for each method to only those that show a negative effect on over 10% of the methods in the training set. The rest of the optimizations maintain their baseline ON/OFF configuration. Figure 9(b) shows the results of this experimental setup. Thus, we can see that the updated heuristic algorithm now achieves average performance that is 8.4% worse than ideal, and only 1.4% worse that the baseline.

There may be several other possible avenues to employ knowledge regarding the behavior and relationships of optimization phases to further improve the performance of online heuristic algorithms. However, both our experiments in this section show the potential and possible directions for improving the effectiveness of existing feature-vector based online algorithms for phase selection.

## 7. Future Work

There are multiple possible directions for future work. First, we will explore the effect of size and range of training data on the feature-vector based solution to the optimization selection problem for dynamic compilers. Second, we will attempt to improve existing heuristic techniques and develop new online approaches to better exploit the potential of optimization selection. In particular, we intend to exploit the observations from Section 4 and focus more on optimizations (and methods) with the most accumulated negative effects to build new and more effective online models. It will also be interesting to explore if more expensive phase selection techniques become attractive for the most important methods in later stages of *tiered* JIT compilers on multi-core machines. Third, we observed that the manner in which some method is optimized can affect the code generated for other program methods. This is an interesting issue whose implications for program optimization are not entirely clear, and we will study this issue in the future. Finally, we plan to repeat this study with other VMs and processor architectures to validate our results and conclusions.

## 8. Conclusions

The objectives of this research were to: (a) analyze and understand the phase selection related behavior of optimization phases in a production-quality JVM, (b) determine the steady-state performance potential of optimization selection, and (c) evaluate the effectiveness of existing feature-vector based heuristic techniques in achieving this performance potential and suggest improvements. We perform our research with the industry-standard Oracle HotSpot JVM to make our results generally and broadly applicable.

We found that most optimization phases in a dynamic JIT compiler only have a small effect on performance, and most phases do not negatively impact program run-time. These experiments also hinted at modest improvements by phase selection in dynamic JIT environments. Correspondingly, the GA-based *ideal* benchmark-wide and per-method optimization phase selection improves performance significantly in a few instances, but the benefits are modest on average (6.2% and 4.3% for per-method and whole-program phase selection customization respectively). This result is not very surprising. To reduce compilation overhead, JIT compilers often only implement the more conservative optimization phases, which results in fewer optimizations and reduced, and possibly more predictable, phase interactions.

We also found that existing feature-vector based techniques used in dynamic compilers are not yet powerful enough to attain the ideal performance. We conducted experiments that demonstrate the directions for improving phase selection heuristics in the future. As part of this research, we have developed the first open-source framework for optimization selection research in a production-quality dynamic compilation environment. In the future, we expect this framework to enable further research to understand and resolve optimization application issues in JIT compilers.

## References

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

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, 2004.

[3] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[4] 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, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, 2006.

[5] J. Cavazos and M. F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the conference on Object-oriented programming systems, languages, and applications*, pages 229–240, 2006.

[6] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizatons. Proc. 2nd Workshop on Feedback Directed Optimization, 1999.

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

[8] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. OBoyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, 2007.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[11] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 180–188, New York, NY, USA, 2005. ACM. ISBN 1-59593-019-1.

[12] K. Hoste and L. Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 165–174, New York, NY, USA, 2008.

[13] http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper 150215.pdf. Memory Management in the Java HotSpot Virtual Machine, April 2006.

[14] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 119–128, 1999.

[15] 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, and T. Nakatani. Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 187–204, 2003.

[16] P. A. Kulkarni. JIT compilation policy for modern machines. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 773–788, 2011.

[17] P. A. Kulkarni, D. B. Whalley, and G. S. Tyson. Evaluating heuristic optimization phase order search algorithms. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–169, 2007.

[18] P. A. Kulkarni, M. R. Jantz, and D. B. Whalley. Improving both the performance benefits and speed of optimization phase sequence searches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, pages 95–104, 2010. ISBN 978-1-60558-953-4.

[19] S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 147–162. ACM, 2012.

[20] H. Lee, D. von Dincklage, A. Diwan, and J. E. B. Moss. Understanding the behavior of compiler optimizations. *Software Practice & Experience*, 36(8):835–844, July 2006.

[21] M. Paleczny, C. Vick, and C. Click. The Java hotspottm server compiler. In *Proceedings of the Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–12, Berkeley, CA, USA, 2001. USENIX.

[22] Z. Pan and R. Eigenmann. PEAK: a fast and effective performance tuning system via compiler optimization orchestration. *ACM Trans. Program. Lang. Syst.*, 30:17:1–17:43, May 2008.

[23] R. Sanchez, J. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. In *Code Generation and Optimization*, pages 257 –266, April 2011.

[24] SPEC98. Specjvm98 benchmarks. http://www.spec.org/jvm98/, 1998.

[25] S. Triantafyllis, M. Vachharajani, N. Vachharajani and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215. IEEE, 2003.