Coupling On-Line and Off-Line Profile Information to Improve Program Performance

Chandra Krintz
Computer Science Department
University of California, Santa Barbara
ckrintz@cs.ucsb.edu

Abstract

Dynamic compilation and optimization are widely used for Internet computing, in which an intermediate form of the code is compiled to native code during execution. An important tradeoff exists between the amount of time spent dynamically optimizing the program and the overall runtime of the program. Since optimization occurs *while* the program is executing, it introduces runtime overhead. For Java programs, this overhead can be substantial due to compilation effort required to convert bytecode to highly efficient machine code.

In this paper, we describe a novel execution environment for Java programs that substantially reduces the overhead due to compilation and optimization by coupling two key dynamic compilation technologies: adaptive optimization and annotation-guided optimization. In the former, profiles are collected using *on-line* instrumentation and sampling of dynamic profile data. In the latter, *off-line* profiles are generated, encoded compactly, and transported with the program files for use by a dynamic optimization system.

Both methodologies (adaptation and annotation), when used alone, have drawbacks: adaptation introduces overhead for on-line instrumentation, measurement, and decision making, and annotations based on off-line profile information can be inaccurate when program inputs for execution and optimization differ from those used for profiling. To combat these drawbacks and to achieve the benefits from both on-line and off-line profiling, we developed a dynamic compilation system (based on JikesRVM) that makes use of both. As a result, we are able improve Java program performance by 9–15% on average, for the programs studied.

1 Introduction

Java programs, which are encoded in an architecture-independent format called bytecode, offer a portable solution to Internet computing. The execution model of bytecode programs in this environment is one in which code and data is first transfered to a client machine and then executed. At the client, the bytecode is converted to native machine code and executed by a virtual machine. Initially, such programs were interpreted. However, to overcome the performance limitations interpretation usually imposes, these systems now employ Just-In-Time compilation [13, 1, 11, 7]. These virtual machines dynamically compile the bytecode stream as each new method is initially invoked. The resulting execution time is lower than for interpreted bytecodes, but execution must pause each time a method is initially invoked so that it may be compiled.

Despite the use of compilation, Java program performance has yet to achieve speeds near that of equivalent C codes. In an effort to reduce this gap, much research has gone into dynamic optimization of Java programs. As a result, various execution environments for

Java [4, 5, 7] now offer extensive optimization techniques. However, due to the algorithmic complexity required to convert bytecode to highly efficient machine code and to the fact that optimization is performed *while* the program is running, performance degradation due to overhead associated with compilation can be significant.

Recent advances in Java virtual machine technology attempt to reduce compilation overhead while maintaining optimized performance levels [2, 5, 7]. These systems apply time-consuming optimizations to only those methods which contribute significantly to overall program performance. Such methods are identified in one of two ways. The first approach is to perform instrumentation and profiling of methods *while* the program is executing [7, 2, 5], i.e. *on-line*. The other approach is to perform *off-line* profiling and to communicate the profile data via annotation [9, 3, 8, 12].

Both on-line and off-line profiling techniques have disadvantages that impact program performance. On-line profiling introduces overhead for instrumentation and measurement sampling. Off-line profiling may be imprecise since, in most cases, the inputs used to execute the program differ from those used during profiling. Hence, decisions based on off-line profiles may be ineffective and possibly degrade performance. Ideally, we want the benefits of both off-line and on-line profiling without the negative side-effects. To this end, we have developed a novel dynamic compilation system for Java programs that uses both on-line and off-line profile data to guide optimization. Our system, on average, improves the overall performance of the programs studied by 9% to 15%.

To implement our system we extended two existing systems: JikesRVM [1], a dynamic and adaptive Java compilation environment and an off-line profiling and annotation system that we developed in prior work [9]. The system that we describe herein uses profile information made available via program annotation to guide method optimization. All unannotated methods are monitored by the on-line profiling system so that profile inaccuracies, if any, do not impact program performance. By coupling the use of profile-based annotations with on-line performance adaptation, we are able to substantially improve program performance.

The contributions that we make with this work include

- A dynamic compilation system for Java that couples on-line and off-line profile data to guide optimization
- Extensions to JikesRVM (a popular, highly optimizing dynamic and adaptive Java execution environment) that enable the use of annotations to guide optimization.
- An empirical evaluation of our system using two profile-aware optimizations. The first uses profiles of the time spent in methods to guide selection of which methods to optimize and at what level. The second uses counts of method calls to guide inlining. The combined use of our resulting optimizations provides significant execution time benefits.
- An empirical analysis of the use of two different types of method profiles to guide optimization decisions: one that is based on program sampling and one that is exhaustive.

In the next section, we detail existing profiling techniques for reducing compilation delay in mobile Java programs. We then describe JikesRVM in Section 3, an existing on-line profile system. In Section 4, we describe how we extended JikesRVM to use off-line profile information. We then present our empirical results in Section 6 and conclude in Section 7.

2 Using Profile Data to Improve Java Program Performance

Since Java programs transfer in a portable, architecture-independent format (bytecode), they must be converted to native machine code prior to execution. Most Java execution environments perform this task by compiling bytecode to native code at the method-level (Just-In-Time). This dynamic compilation occurs *while* the program is running and as such, it introduces intermittent program interruption (and possibly an overall degradation in performance). These delays are compounded when optimization is used to reduce execution time.

The Java bytecode format is fundamentally time consuming to compile and optimize. The format is a 0-address, or stack-machine, representation in which operations are performed using a single stack data structure. Since most of the hardware architectures in use today implement a register machine model, bytecode programs must be translated from the stack model to the register model. This is not a straight-forward transformation since requires symbolic execution (similar to that used for verification of type safety [10]) and highly complex optimization algorithms to enable high-performance.

One common methodology for reducing compilation overhead is to only apply optimization to the parts of the program that impact overall program performance. However, to identify these program pieces (which are in our case, methods), we must execute the program and gather information on, i.e., profile, the execution characteristics of the program.

Dynamic (Just-In-Time) Java compilation systems use profile information in exactly this way to reduce the overhead of optimization and yet enable optimized execution speeds. Most commonly, these systems profile the program *on-line*, while it is executing. This profile information is then used to select methods for optimization and to guide the amount of optimization that is applied. Examples of such systems include HotSpot [7] from Sun Microsystems, JikesRVM [1] from IBM, and the Open Runtime Platform (ORP) [5] from

Intel Corporation. An alternate solution is to profile the program *off-line* and communicate the necessary information to the dynamic compilation system with the program. We have developed such a system in prior work [9]. We describe both types of systems in the following subsections.

2.1 On-Line Profiling

On-line profiling is implemented by inserting instrumentation into the native machine code of methods during compilation (immediately prior to execution). The instrumentation causes profile information about the program's execution to be logged for use by a compilation system. Program characteristics that are commonly profiled by existing systems include counts of method invocations, back-edges in the program control flow graph that are taken (which indicates looping), and edges of the program method call graph that are traversed (which identifies frequented call paths through the program).

When a method is first invoked, it is compiled without optimization (fast-compiled) since the system has not yet gathered profile information on the method. Once accumulated profile information on the method indicates that the method is a significant contributor to overall program performance, the method is re-compiled using optimization. Any number of optimizations can be performed on the method upon re-compilation; most compilers group optimizations into *levels* and perform all of the optimizations in that level. In addition, the system may again insert instrumentation during re-compilation so that increasingly higher levels of optimization can be applied if the method's execution characteristics indicate that it is necessary. This process is called adaptive optimization.

Such systems reduce compilation overhead with little loss in program performance since they expend optimization effort only for those methods that account for a significant portion of execution time. The primary drawback to such adaptive systems is that the instrumentation, measurement, logging, and processing of dynamically-changing profile information occurs at runtime and hence, imposes overhead that degrades program performance.

2.2 Off-Line Profiling

An alternate solution to compilation overhead that does not require the overhead of on-line measurement is *off-line* profiling. Off-line profile tools can add pertinent profile information to the program files so that it is communicated to a dynamic compilation system when the program executes. Using off-line profiles has the added benefit that the profiles need to be generated only a single time as opposed to each time the program is executed, as in the on-line case.

In prior work, we developed an off-line profiling system which used program annotations (as Java class file attributes) to communicate profile data [9]. Annotations have also been used to guide dynamic optimizations that are not profile-based [12, 3, 8], i.e., the annotations communicate static analysis information to the dynamic compilation system for use in optimization. Examples of such annotation-based optimizations include register allocation [3, 8], array bounds check and null pointer check elimination [12], escape analysis [6]. With this work, we focus solely on profile-based annotations.

Off-line profiling techniques avoid the overhead of on-line instrumentation and measurement but have the disadvantage of profile inaccuracy across program inputs. That is, if one set of inputs is used to generate profile data off-line and another set is used for program execution, profile-guided decisions may be incorrect and hence, be ineffective or possibly cause performance degradation. On-line adaptive systems do not have this problem since profile information is gathered as the program is executing.

To achieve the benefits from on-line profiling (accurate profile information) *and* off-line profiling (reduced on-line overhead) without the associated disadvantages, we developed a system which uses *both* to reduce compilation delay and to improve program performance over either constituent methodology alone. To enable this, we extended an existing on-line profiling system, JikesRVM [1], so that off-line profile information is also used to guide optimization decisions.

3 JikesRVM

JikesRVM [1] is an open-source, dynamic and adaptive optimization system for Java from IBM T.J. Watson Research Center. JikesRVM was designed and continues to evolve with the goal of enabling high-performance. JikesRVM compiles (at runtime) Java bytecode programs at the method-level, Just-In-Time, to x86 (or PowerPC) code. The system, in addition to extensive runtime services (garbage collection, thread scheduling, synchronization, etc.), implements adaptation through the use of an on-line instrumentation and profile collection facility called the Controller [2]. The Controller uses program execution characteristics collected *on-line* to determine where to expend optimization in an effort to reduce compilation overhead.

JikesRVM implements two compilers to facilitate re-compilation (adaptation): One com-

Program	Baseline TT (CT)	Opt TT (CT)	Adapt TT (CT)
_201_compress	52.90 (0.19)	8.85 (0.19)	10.54 (0.25)
_202_jess	18.04 (0.60)	6.33 (0.52)	5.63 (0.68)
_209_db	21.62 (0.20)	17.81 (0.19)	15.86 (0.25)
_213_javac	13.64 (0.94)	13.43 (0.79)	10.34 (0.79)
_222_mpegaudio	60.01 (0.64)	8.08 (0.30)	8.54 (0.49)
_227_mtrt	14.60 (0.34)	6.37 (0.25)	6.24 (0.42)
_228_jack	23.89 (0.43)	8.96 (0.30)	5.65 (0.51)
Average	29.24 (0.44)	9.98 (0.36)	8.97 (0.48)

Table 1: JikesRVM performance characteristics. The first column shows the total time (TT) and compile time (CT) in parentheses, required for benchmark execution when only the Baseline compiler is used. Total time is compile plus execution time. The second column shows the same statistics for the Optimizing compiler. The final column is the total time and compile time for the adaptive system in which only those methods that are identified as "hot" by the on-line profiling system are optimized.

piler (Opt) is highly-optimizing but slow and the other is non-optimizing (Baseline) and very fast. Various JikesRVM performance characteristics are shown in Table 1. The first column shows the total time (TT) and compile time (CT) in parentheses required for benchmark execution when only the Baseline compiler is used. Total time here is compile plus execution time. The second column shows the same statistics for the Optimizing compiler. The final column is the total time and compile time for the adaptive system in which only those methods that are identified as "hot" by the Controller are optimized. We provide further details and statistics about the benchmarks and our execution environment in our experimental methodology section (Section 5). On average, Baseline compilation time is 1.44 times faster than Opt and Baseline execution time (without compilation) is 3.27 times slower than Opt for these benchmarks.

Upon initial invocation using JikesRVM, methods are Baseline-compiled (without optimization). Both Opt and Baseline compilers instrument method code during compilation (as deemed necessary by the Controller) so that performance data is measured and logged while the program executes. When measurement values exceed certain thresholds, the Controller initiates method re-compilation (by the Opt compiler) using increasing levels of optimization. Currently, the Opt compiler has three levels of optimization: 0, 1, and 2. This process continues until the method is fully optimized.

The Controller measures the time spent in methods and the number of method invocations each method executes at each call site. The former is used to guide re-compilation using additional optimization since it identifies methods that account for a significant portion of program execution time. To measure the time spent in a method, the Controller counts method invocations and taken backedges (loop iterations). Method call counts are used to guide re-compilation of methods so that inlining of methods can be performed, i.e., adaptive inlining.

Measurements are made *only* when thread yield points are taken, i.e., measurement is sample-based and not exhaustive. Thread yield points are places in the code at which the currently running thread yields control of the processor to other threads (including JikesRVM threads, e.g., the Controller, the thread scheduler, etc.) In addition, the logged values are sampled at an empirically selected sampling frequency and decayed to account for changes in execution characteristics over the program's lifetime. Each of these optimizations reduce the accuracy of the profile information while reducing the overhead of on-line measurement. Additional information about the JikesRVM Controller and adaptive optimization system can be found in [2].

4 Combining Off-Line and On-Line Profile Data

To implement the use of both on-line and off-line profiles, we extended JikesRVM with the profile-based annotation system that we developed in prior work [9]. The previously developed system consists of an annotation insertion tool (AIT) and an annotation-guided optimization system built into the Open Runtime Platform (ORP) from Intel Corporation. For this work, we implemented the ORP extensions in JikesRVM. We selected JikesRVM for this study due to its extensive implementation of optimization and adaptation.

To evaluate the efficacy of combining off-line and on-line profile data for use in optimization we implemented two annotation-based optimizations in JikesRVM: *Hot Method Optimization* and *Hot Call-site Inlining*. In the former, annotated methods indicate that the method should be optimized; the annotation also contains the optimization level that should be used. Hot call-site inlining is an annotation-based optimization in which method call-site annotations identify context-free inlining opportunities.

We first extended the AIT to profile the time spent in methods so that "hot" methods in the program could be annotated. AIT previously implemented this method selection by counting the number of method invocations that are performed by a given Java program. We extended this mechanism to consider method invocations as well as back-edges taken. In addition, we gathered two different types of hot method profiles. The first uses the JikesRVM Controller system (in which counts of method invocations and backedges are made only when thread yield points are taken). The second uses the same mechanism but counts every method invocation and back-edge taken, exhaustively. We gathered these two different profiles so that we could evaluate and compare the efficacy of each.

In addition, we extended AIT to enable automatic annotation of JikesRVM adaptive inlining decisions [2]. During off-line execution, each call site at which context-free inlining is performed by JikesRVM is annotated by the AIT so that the same profiling can be performed without the overhead of on-line method call profiling.

The JikesRVM extensions that we implemented enable the system to read, decompress, and process compressed annotations which are included in Java class files as class file attributes [9, 10]. The resulting system is one that uses off-line profile data whenever possible to avoiding the overhead of on-line instrumentation and measurement. Upon program invocation, annotated Java programs are dynamically loaded from the source by the execution site. During class loading, annotation data is decompressed and stored for use by the optimization system. When a method is marked with an annotation, it is optimized at the annotated optimization level when it is first invoked. During compilation, if the Opt compiler encounters an annotated call site, it inlines the called method. All unannotated methods are then instrumented and monitored by the existing JikesRVM Controller system.

5 Experimental Methodology

The results we present were gathered by repeatedly executing benchmark programs on a dedicated 2.4Ghz x86-based single-processor Xeon machine (with hyperthreading enabled) running Debian Linux v2.4.18. In addition, we used JikesRVM version 2.1.1 build 9-12-02 with jlibraries 08-14-19-54-59.

The applications we examine are the SpecJVM benchmark programs. We report results using both the large input (100) and the small input (10). General statistics on the bench-

Program	Class Count	Method Count	Used Methods 100 (10)	ADAPT HOT 100 (10)
_201_compress	12	44	32 (32)	16 (10)
_202_jess	151	690	412 (412)	45 (10)
_209_db	3	34	27 (24)	7 (1)
_213_javac	176	1190	740 (713)	124 (15)
_222_mpegaudio	55	322	201 (200)	75 (34)
_227_mtrt	26	180	82 (81)	59 (18)
_228_jack	56	315	265 (265)	35 (8)
Average	68	396	251 (247)	52 (14)

Table 2: Benchmark statistics. The first two columns show the number of static classes and methods, respectively contained in each benchmark. The third column is the number of methods executed by each program using the size 100 input and size 10 input (in parentheses). The last column shows the number of methods that are considered "hot" by the JikesRVM adaptive system for both inputs (100 and 10 (in parentheses)).

marks are shown in Table 2. The first two columns show the number of static classes and methods, respectively, contained in each benchmark. The third column is the number of methods executed by each program using the large, size 100, input and small, size 10, input (in parentheses). The last column shows the number of methods that are considered "hot" and hence optimized using the Opt compiler, by the JikesRVM adaptive system during execution; all other methods are Baseline-compiled (without optimization).

Since the inputs used to collect an off-line profile may be different from those used when execution (and optimization) of the program eventually occurs, we measured the efficacy of our techniques for both cases. When the input (in our case, 100) is the same for both off-line profiling and result generation, we refer to the results as *same-input* and distinguish them in our graphs and tables using the *Same* keyword. *Cross-input* results are those in which we use a profile generated using the small, size 10 input then execute and optimize using the large, size 100 input. These results in our graphs are distinguished using the *Cross* keyword.

Motivating Example: On-line and Off-line Profiling Alone			
	On-Line	Off-line (Secs)	
Program	(Secs)	Same (%imp)	Cross (%imp)
_201_compress	10.54	7.30 (31%)	7.48 (29%)
_202_jess	5.63	4.31 (23%)	12.12 (-115%)
_209_db	15.86	16.56 (-4%)	18.94 (-19%)
_213_javac	10.34	9.47 (8%)	15.05 (-45%)
_222_mpegaudio	8.54	7.77 (9%)	11.17 (-31%)
_227_mtrt	6.24	5.16 (17%)	6.56 (-5%)
_228_jack	5.65	5.39 (5%)	6.16 (-9%)
Average	8.97	7.99 (13%)	11.07 (-28%)

Table 3: Total time (compilation plus execution) in seconds for each benchmark resulting the use of profiles gathered on-line (column 1) and off-line (columns 2 and 3) to guide selection of methods to optimize. Since the inputs used for off-line profiling may be different than the inputs used to execute (and hence, optimize) the program, we include results for both cases. *Same* indicates that the same input is used for profile and result generation (in this case, size 100); *Cross* indicates that a different input is used to generate the profile (in this case, size 10). We use the size 100 input for result generation in both cases, regardless of profile input. For each of the off-line measurements, we also include the percent improvement over the use of on-line profiling alone. Percentages greater than 10 identify cases in which off-line profiling degrades performance. The percent improvement shown for the average of the Same column results from the overhead imposed by on-line instrumentation and measurement.

6 Results

We first motivate the problem we are addressing with this work by presenting performance data on adaptive (on-line) optimization and off-line optimization. Table 1 shows the total time (compilation plus execution) in seconds for each benchmark using each constituent technology alone: On-Line profiling (adaptation) and off-line profiling (annotation-based). The first column of data shows total time using the JikesRVM adaptive optimization system without off-line profile information. On average, the programs execute in 9.00 seconds.

The last two columns of data show the total time using off-line annotations alone to identify methods for optimization. The "Same" column shows the total time when the same input is used to profile and annotate the program as is used for result generation. This is the perfect information case. We generated this off-line profile using the JikesRVM adaptive system, i.e., we recorded the JikesRVM decisions about which methods to optimize (and at what level) and annotated the program with this information. Therefore, both the on-line and off-line system (first and second columns, respectively) use the same profile to guide optimization. On average, the programs execute in 7.99 seconds, 13% faster than the on-line system. This difference indicates the overhead introduced by JikesRVM for on-line measurement and profile analysis.

The "Cross" column shows the total time when the inputs used for profile generation differ from those used to execute (and hence, optimize) the program. For all of our cross-input results, we use the profile generated using the JikesRVM Controller and the small, size 10, input. We then executed each program using the large, size 100, input; hence, optimization is guided by size 10 input profile information. In all but one case, inaccurate, off-line profile data degrades performance. On average, cross-input profiling is 28% slower than the JikesRVM adaptive system. In one case, off-line same-input profile data degrades performance (_209_db), we revisit this problem later in this section.

This table exemplifies the problem we are attempting to solve: On-line profiling introduces unnecessary overhead due to instrumentation and measurement and off-line profiling can be ineffective or cause performance degradation. This set of experiments motivated us to combine off-line and on-line profiling to exploit the potential from both in an effort to improve Java program performance.

Once we implemented "annotation-awareness" in JikesRVM, we experimented with two

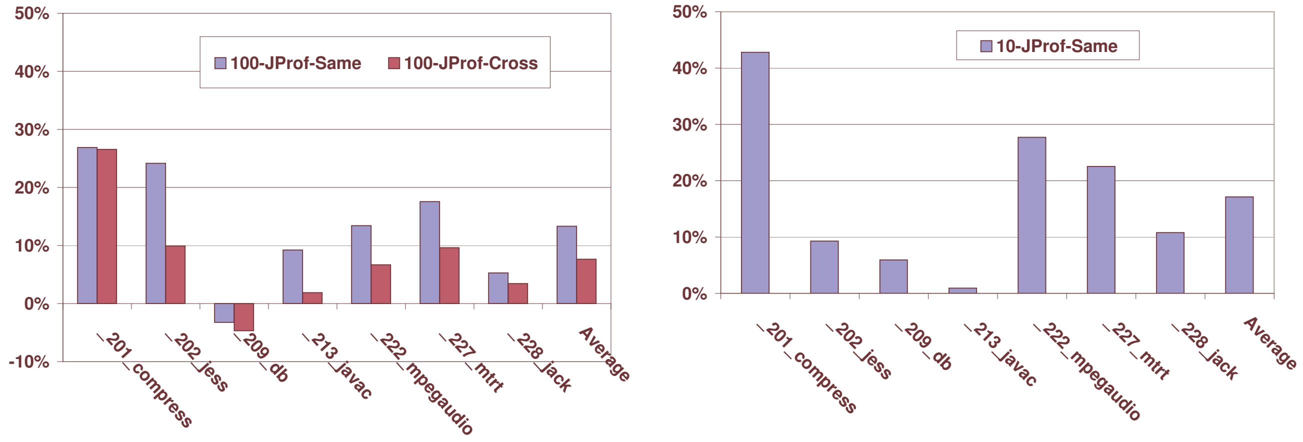

Figure 1: The percent improvement over JikesRVM (on-line profiling alone) that results when on-line and off-line profile data is combined to guide selection of methods for optimization. The left graph is the improvement for the size 100 inputs and the right graph is the size 10 inputs. The left graph includes data for both same-input and cross-input results.

different types of profiles to guide selection of methods to optimize (the Hot Method Optimization). The first type of profile is one (as mentioned above) in which we use JikesRVM itself, off-line, for profile generation. We refer to this profile type as *JProf* in our tables, graphs, and text. As described above, method invocations and taken backedges are only counted when the executing thread yields the processor while in a method prologue or on a loop backedge, respectively. We annotated each hot method with optimization level of 1 since it is level that is most commonly selected by the the JikesRVM adaptive system. We plan to consider the efficacy of annotating different levels as well as different individual optimizations as part of future work.

Figure 1 shows the percent improvement enabled by our JikesRVM extension in which off-line profiles are used in conjunction with those generated on-line. The base performance against which we are comparing is JikesRVM adaptive compilation (the "On-Line" column in Table 3). The right graph are the results for the size 100 input (which shows same-input and cross-input results). The left graph is for the size 10 input (same-input only). For all but one benchmark, combining off-line and on-line profiling data to guide method selection and optimization, substantially improves performance: On average, we improve execution time by 8% across inputs. This is a significant improvement over both the on-line or off-line system alone (the data for which was shown previously in Table 3).

The benchmark for which we degrade performance is _209_db. The reason for this degradation is due to missed inlining opportunities. We address this problem with our second optimization (Hot Call Site Inlining) below.

The second type of profile we investigated is one in which method invocation and loop backedge counts are made exhaustively (as opposed to only at thread yield points). We refer to this type of profile as *EProf*. We gathered performance results for EProfs in an effort to identify and compare the performance impact of non-exhaustive (JProfs) with exhaustive measurement (EProfs).

To generate the EProfs, we measured the total time spent in each method executed (method invocation counts plus backedges taken). We then ordered these profiles in decreasing order so that the start of the file contained the "hottest" methods: those methods that contributed most to overall execution time.

We then selected various percentages of hot methods and measured the efficacy of their use as annotations to guide the selection of methods to optimize. We studied a wide range of percentages and present 5%, 10%, 15%, 20%, 25%, 30%, 40%, and 50% for brevity. Higher percentages only degraded performance. Each percentage indicates the percentage of hot methods that we selected from the profile data, e.g., for 5% we annotate the top 5% of the

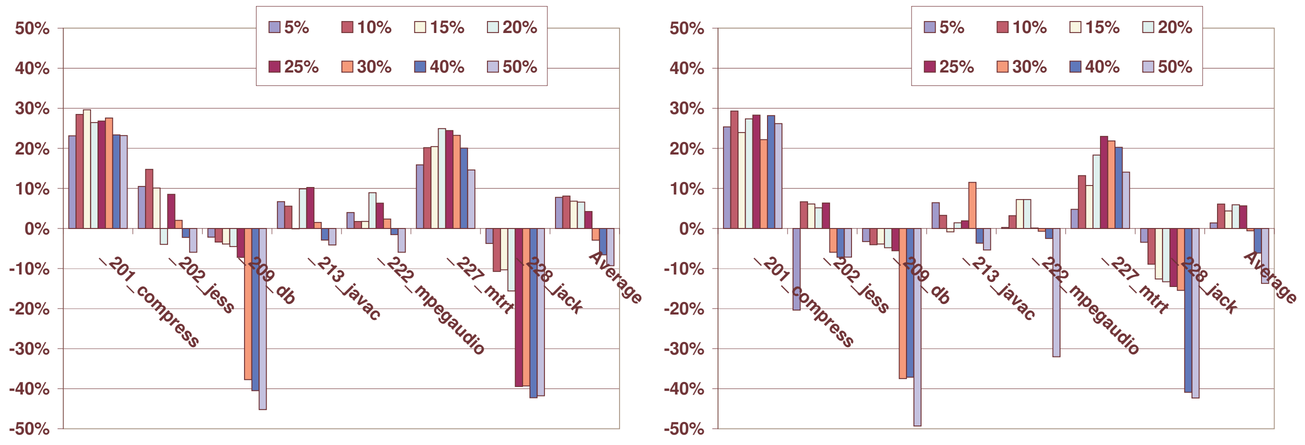

Figure 2: Percent improvement (or degradation) in performance (over JikesRVM - on-line profiling alone) using exhaustive profiles (EProfs) and various percentages of hot method selection. The left graph are the same-input results and the right graph are those for cross-input. We used this data to select percentages that perform best for each program across all configurations (Size 100 same-input, Size 100 cross-input, and Size 10 same-input).

hot methods so that only these methods are optimized. Again, for each hot method we annotated an optimization level of 1 since it is the most prominent level selected by the JikesRVM adaptive system.

The graphs in Figure 2 show our same-input (left graph) and cross-input (right graph) results, respectively for this series of experiments. There is a wide variation in performance for the different hot percentages. The averages shown at the far right of the graphs indicate the performance trend. On average, by optimizing the top 10%, top 15%, top 20%, or top 25% of hot methods results in improved performance over the base system (JikesRVM), on average. All of this data is for program execution using the size 100 input. We have left out the size 10 results due to space constraints. However, the size 10 results exhibit similar behavior.

To compare these results to JProf, we first selected the EProf percentage that performed

	JProf		Best EProf		Best EProf
Program	Same	Cross	Same	Cross	Best%
_201_compress	27%	27%	28%	29%	10%
_202_jess	24%	10%	15%	7%	10%
_209_db	-3%	-5%	-2%	-3%	5%
_213_javac	9%	2%	10%	2%	25%
_222_mpegaudio	13%	7%	9%	7%	20%
_227_mtrt	18%	10%	25%	18%	20%
_228_jack	5%	3%	-4%	-3%	5%
Average	13%	8%	12%	8%	14%

10-JProf	Best EProf
43%	35%
9%	-25%
6%	7%
1%	-120%
28%	31%
23%	5%
11%	-20%
17%	-13%

Table 4: Percent improvement over JikesRVM (on-line profiling alone) when JikesRVM taken-yield-point counters (JProfs) and exhaustive counts (EProfs) are used during profile generation. On average, there is little difference in performance between the two types of counters. The left table are the size 100 input results and the right table show the size 10 table results. For short running, programs (size 10), EProfs can significantly degrade performance. As such, JProfs are a better choice for our system.

best for each program over all configurations (Size 100 same-input, Size 100 cross-input, and Size 10 same-input). Table 4 shows this selected percentage (in the final column) as well as provides a comparison between the use of JProfs and EProfs. The first two columns (JProf) indicates the percent improvement of our system over JikesRVM alone when JProfs are used to annotate methods for optimization. This is the same data that we show in the left graph of Figure 1. The next two columns (Best EProf) show the percent improvement that results from the use of the best-performing EProf percentage. The left table shows size 100 input results and the right table shows size 10 input results.

On average, there is little difference between JProfs and EProfs for input size 100. However, for short running programs (size 10), EProfs can significantly degrade performance. As such, JProfs are a better choice for our system.

We next consider the effect of our second optimization: Hot Call Site Inlining. For

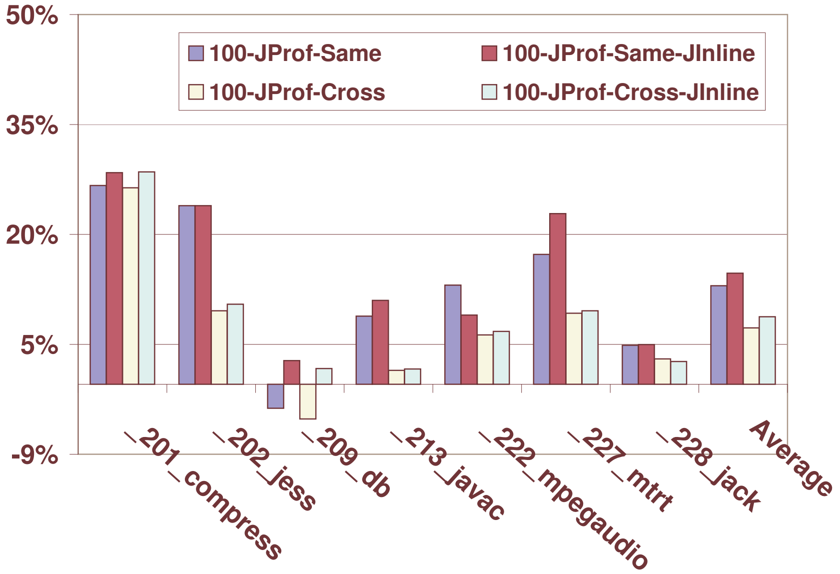

Figure 3: The effect of combining off-line and on-line profile data to guide method optimization *and* context-free inlining. The data is the percent improvement over the JikesRVM adaptive system for input 100. The first and third bars are the same as those presented previously in Figure 1: the same-input and cross-input results using the JProf off-line profile to guide method optimization. The second and fourth bars show the combined effect of JProf for method optimization and JInline for inlining, respectively. For all but one benchmark (_209_db) the combined use JInline and JProf profile is substantially less than when JProf is used in isolation. For this benchmark, however, performance degradation is eliminated. We are currently investigating ways in which we can identify when the use of JInline profile data will be effective.

this optimization, we profile context-free inlining decisions made by the JikesRVM adaptive system, off-line. We then use this profile, which we refer to as JInline, to annotate the call sites that should be inlined.

Our results are shown in Figure 3 as percent improvement over the JikesRVM adaptive system (online-only profiling). The JProf results are repeated from Figure 1; we provide them again for reference. These JProf results are shown in the first and third bars of data for each benchmark. The data for the combined use of JProf and JInline profiles is shown as the second and fourth bars for each benchmark. We have left out the data for input 10 since there are no annotated call sites when input 10 is used for any program except _209_db (for which relative improvements are similar to the size 100 data for this program).

By using both profiles, i.e., both optimizations (Hot Method Optimization and Hot Call Site Inlining), we achieve additional performance gains over using JProf in isolation. In addition, we eliminate the _209_db performance degradation that occurs when we use JProfs alone. The only case in which our system does not experience additional performance gains due to the use of JInlines is for _222_mpegaudio (using the same inputs for profile and result generation). We believe that this is due to incorrect JikesRVM selection of call sites to inline. However, we are continuing to investigate this anomaly. On average, our system, using JProfs and JInlines collected off-line, in conjunction with the JikesRVM on-line Controller system, is able to improve Java program performance by 15% using same-input profiles and 9% across inputs.

7 Conclusions

We have presented a novel execution environment for Java programs substantially improves program performance by coupling two key dynamic compilation technologies: adaptive optimization and annotation-guided optimization. In the former, on-line profiles are collected using instrumentation and sampling of dynamic profile data. In the latter, off-line profiles are generated, encoded compactly, and transported with the program files for use by a dynamic optimization system.

Both methodologies (adaptation and annotation), when used alone, have drawbacks: adaptation introduces on-line overhead for instrumentation, measurement, and decision making, and annotations based on off-line profile information can be inaccurate when program inputs for execution and optimization differ from those used for profiling. To combat these drawbacks and achieve the benefits from the use of on-line *and* off-line profile data, we have developed a system that makes use of both.

Our system is an extension of the JikesRVM execution environment for Java programs from IBM T.J.Watson Research Center. We have extended it to accept bytecode annotations and to use the profile data encoded in the annotations to guide optimization decisions. When the off-line profile data is inaccurate, on-line profiling is performed. By coupling both types of profile data, we are able to reduce on-line overhead and thus improve overall program performance by 9-15% for the programs studied.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[2] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the jalapeño jvm. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2000.

[3] A. Azevedo, A. Nicolau, and J. Hummel. Java Annotation-Aware Just-In-Time Compilation System. In *ACM Java Grande Conference*, pages 142–151, June 1999.

[4] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, pages 129–141, June 1999.

[5] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[6] M. Franz, V. Haldar, C. Krintz, and C. Stork. Online Verification of Offline Escape Analysis. Technical Report 2002-26, University of California, Santa Barbara, Sep 2002.

[7] Sun Microsystems Inc. The Java Hotspot Virtual Machine White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_%4_30_01.html`.

[8] J. Jones and S. Kamin. Annotating Java Class Files with Virtual Registers for Performance. *Journal of Concurrency: Practice and Experience*, 12(6):389–406, May 2000.

[9] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 156–167, June 2001.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[11] Open Runtime Platform (orp) from Intel Corporation. `http://intel.com/research/mrl/orp`.

[12] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. In *Proceedings of IBM Centre for Advanced Studies Conference (CASCON)*, pages 152–168, 2000. `http://www.sable.mcgill.ca/publications`.

[13] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.