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

In this paper, we describe a novel execution environment for Java programs that substantially improves execution performance by incorporating both on-line and off-line profile information to guide dynamic optimization. By using both types of profile collection techniques, we are able to exploit the strengths of each constituent approach: profile accuracy and low overhead.

Such coupling also reduces the negative impact of these approaches when each is used in isolation. On-line profiling introduces overhead for dynamic instrumentation, measurement, and decision making. 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% 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 [16, 1, 13, 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 [5, 6, 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, 6, 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, 6], i.e. *on-line*. The other approach is to perform *off-line* profiling and to communicate the profile data via annotation [9, 14].

Both on-line and off-line profiling techniques have disadvantages that impact program performance. On-line profiling (used in adaptive optimization systems) introduces overhead for instrumentation and measurement sampling. Off-line profile data 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 profiling may be ineffective and can possibly degrade performance.

Ideally, we want the benefits of both on-line and off-line profiling (profile accuracy and minimal on-line overhead) without the negative side-effects. To this end, we have extended the JikesRVM [1], an adaptive optimization system to also consider off-line profile information to guide optimization. We enable communication of off-line profile information to the JikesRVM using bytecode annotation [9]. By combining annotation (off-line profiling) with adaptive optimization (on-line profiling), we are able to achieve performance levels that exceed that of either constituent mechanism in isolation.

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. The use of the former reduces the overhead of on-line instrumentation and measurement. The use of the latter compensates for profile inaccuracies across inputs.
- Extensions to JikesRVM (a popular, highly optimizing dynamic and adaptive Java execution environment) that enable the use of annotations to guide optimization.
- An analysis of the efficacy of sampled and exhaustive measurement for off-line profile collection. We find that there is little difference in performance when sampled data is used in place of exhaustively collected data. This confirms the findings in similar studies [3].
- 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 context-free inlining. The combined use of our resulting optimizations, on average, improves the overall performance of the programs studied by 9%.

In the next section, we describe the strengths and weaknesses of existing (on-line and off-line) profile-based techniques for reducing compilation delay in Java programs. In Section 3, we describe how we extended JikesRVM to also use off-line profile information. We then detail our methodology (Section 4), empirical results (Section 5), and the related work (Section 6). Finally, we present our conclusions 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 straightforward transformation since it requires symbolic execution (similar to that used for verification of type safety [12]) and highly complex optimization algorithms to enable high-performance.

To reduce the overhead required for dynamic compilation and optimization of Java programs, many execution environments apply optimization to only those 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. The profile information is then used to select methods for optimization and to guide the amount of optimization that is applied. Profile generation can be generated on-line, *while* the program is running or off-line as part of the program development phase (prior to execution by users).

2.1. On-Line Profiling

On-line profiling is implemented by inserting instrumentation into the native machine code of methods during dynamic compilation. 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 or interpreted) since the system has not yet gathered profile information on the method. When accumulated method profile information indicates that the method is a significant contributor to overall program performance, the method is re-compiled using optimization. Any level of optimization can be performed on the method upon re-compilation. 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*.

Examples of adaptive optimization systems include HotSpot [7] from Sun Microsystems, JikesRVM [1] from IBM, and the Open Runtime Platform (ORP) [6] from Intel Corporation. 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.

There are two performance drawbacks to such adaptive systems. The first is that the instrumentation, measurement, logging, and processing of dynamically-changing profile information occurs at runtime and hence, each imposes overhead that degrades program performance. The second drawback is *execution overhead*, i.e., the optimization opportunity lost, due to the delay between when a hot method is first invoked and when it is identified by the on-line measurement system as hot. Each invocation of a hot method invoked using unoptimized code reduces performance potential of adaptive optimization.

JikesRVM from IBM T. J. Watson Research center is the on-line profiling system that we investigate in this study. JikesRVM, uses a sampling-based approach to on-line measurement to reduce the overhead of measurement and analysis. Measurements are made *only* when a thread yields the processor to other threads. at an empirically selected sampling frequency. Sampling is able to reduce some on-line overhead; however, as we will show, additional improvements are still possible and necessary.

The performance characteristics of the JikesRVM adaptive optimization system on a 2.4GHz x86 Xeon processor 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 non-optimizing (Baseline) compiler is used. TT is compile plus execution time. The second column shows the same statistics for the optimizing (Opt) compiler. The final column (Adapt) is the total time and compile time for the adaptive system in which only those methods that are identified as “hot” by the system are optimized with Opt; all others are Baseline compiled. The benchmarks are taken from SpecJVM98 [15] and this data was collected using the size 100 input. We provide further details and statistics about the benchmarks and our execution environment in Section 4.

On average, JikesRVM 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. Adaptive optimization posts larger compilation times than Opt since methods that remain hot are optimized multiple times using higher levels of optimization. This results in lower overall total times (execution plus compilation time); the programs on average execute in 9.00 seconds. On average, 20

	Baseline	Opt	Adapt
Program	TT (CT)	TT (CT)	TT (CT)
compress	52.90 (0.19)	8.85 (0.19)	10.54 (0.25)
db	21.62 (0.20)	17.81 (0.19)	15.86 (0.25)
jack	23.89 (0.43)	8.96 (0.30)	5.65 (0.51)
javac	13.64 (0.94)	13.43 (0.79)	10.34 (0.79)
jess	18.04 (0.60)	6.33 (0.52)	5.63 (0.68)
mpeg	60.01 (0.64)	8.08 (0.30)	8.54 (0.49)
mtrt	14.60 (0.34)	6.37 (0.25)	6.24 (0.42)
AVG	29.24 (0.44)	9.98 (0.36)	8.97 (0.48)

Table 1. JikesRVM performance characteristics. The first two columns show the total (compile plus execution) time (TT) and compile time (CT), in parentheses, for each SpecJVM98 program (using size 100 input) when only the non-optimizing (Baseline) compiler or the optimizing (Opt) compiler is used, respectively. The final column (Adapt) 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 (20% on average) are optimized.

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.

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.

Table 2 shows the performance characteristics that result from using off-line profile information. The “Same” column shows the total (compile plus execution) 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. The parenthesized values in the table show the percent improvement over JikesRVM (on-line) adaptive system. 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 (Table 1 “Adapt” column and Table 2, 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

	Off-line (Secs)	
Program	Same (%imp)	Cross (%imp)
compress	7.30 (31%)	7.48 (29%)
db	16.56 (-4%)	18.94 (-19%)
jack	5.39 (5%)	6.16 (-9%)
javac	9.47 (8%)	15.05 (-45%)
jess	4.31 (23%)	12.12 (-115%)
mpeg	7.77 (9%)	11.17 (-31%)
mtrt	5.16 (17%)	6.56 (-5%)
AVG	7.99 (13%)	11.07 (-28%)

Table 2. Total time (compilation plus execution) in seconds for each benchmark resulting the use of profiles gathered off-line to guide selection of methods to optimize. *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. In parentheses, we show the percent improvement (or degradation) over on-line profiling (the JikesRVM adaptive system – "Adapt" column in Table 1).**

is the overhead introduced by JikesRVM for on-line measurement and profile analysis plus execution overhead, i.e., slower execution times due to lost optimization opportunity.

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 again used JikesRVM for profile generation using input size 10. We then executed each program using the size 100 during which optimization is guided by the size 10 input profile information. We show the percent improvement (or degradation) over JikesRVM adaptive optimization, in parenthesis. In all but one case, inaccurate, off-line profile data degrades performance. In analyzing the off-line profile data, we found that the off-line profiles omitted many of the hot methods and as such, much of the execution was unoptimized. On average, cross-input profiling is 28% slower than the JikesRVM adaptive system.

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 due to cross-input profile inaccuracies. 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.

3. Combining Off-Line and On-Line Profile Data

To implement the use of both on-line and off-line profiles, we extended JikesRVM, an open-source, dynamic and adaptive (on-line profiling) optimization system for Java developed by 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 on-line instrumentation and profile collection and evaluation to determine where to expend optimization in an effort to reduce compilation overhead.

We extended JikesRVM with an off-line profile system that we developed in prior work in which program characteristics are collected off-line and communicated to the dynamic compilation system via bytecode annotation [9]. The annotated information is used by the compilation system to guide optimization. We encode each annotation and compress it using zlib compression to ensure that any transfer delay overhead that we introduce is negligible. We provide actual measurements of annotation size in our description (below) of each annotation-based optimization that we implemented.

We implemented two annotation-based optimizations in JikesRVM to evaluate the efficacy of using both off-line and on-line profile information to guide optimization. The first is *Hot Method Optimization (HMO)* in which annotated methods indicate that the method should be optimized; all un-annotated methods are Baseline compiled. The second optimization we implemented is *Hot call-site inlining (HCI)* in which call-sites that should be inlined are annotated. For both optimizations, we generated two profiles each for a different program input.

For HMO, we generated profiles using two different methods: sampling and exhaustive measurement. We performed this study to determine whether sampled data accurately reflected exhaustive measurement for hot method identification and to evaluate the performance of each technique for cross-input optimization. Since off-line profile information is input dependent, we wanted to investigate which technique enabled better performance across inputs. In the results section, we refer to sampled profile data as *JProf* and exhaustive profile data as *EProf*.

To collect profile data, we instrumented the JikesRVM adaptive system (executed off-line) and logged its decisions during program execution. To identify hot methods, JikesRVM estimates the time spent in methods by counting method invocations and taken backedges (loop iterations). 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 system threads). 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. For JProf data, we simply collected the JikesRVM optimization and inlining decisions for HMO and HCI annotation data.

For collection of EProf data, we modified JikesRVM to take measurements at method invocations and taken backedges regardless of whether a thread yield point was taken. We then ordered this data by highest total count and generated annotations using different top (hottest) percentages of methods. For example, we annotated the top 25% of hottest methods to evaluate its impact on performance. We evaluated a wide range of percentage values.

Each annotation inserted into class files is a tuple that consists of annotation opcode, size, and annotation data, e.g., $< opcode, size, data >$. Both *opcode* and *size* are each two bytes in length and *data* has a variable length that is indicated by *size*. All annotations are collectively compressed and inserted into the class file attribute [12] of the appropriate class using an annotation insertion tool that we developed.

For HMO, we inserted a annotations that contained the method identifiers (starting at 0 and increasing in the order the methods appear in the class file) for the annotation data. For example in the annotation $< 0, 4, 1\ 0\ 3\ 1 >$, 0 indicates HMO, 4 is the number of bytes that follow for the annotation data, 1 and 3 indicate that the second and fourth methods should be optimized at optimization levels 0 and 1, respectively. In the annotation $< 1, 6, 0\ 4\ 4\ 32 >$, 1 indicates HCI, 6 is the number of bytes for the annotation data, 0 identifies the first method, 4 is the number of bytes that follow for this method, 4, and 32 indicate bytecode instructions in method 0 that are call-sites that should be inlined (are hot). The instruction identifiers are each two bytes in length; all other entries are one byte. On average across benchmarks using both annotations, we add 80 bytes per application (profiled using the size 100 input).

We also extended JikesRVM to read, decompress, and process compressed annotations included in Java class files. Each time a class is loaded the annotations are extracted and stored in a hash table for use during optimization. Given HMO and HCI annotations, our version of JikesRVM checks this hash table when it initially compiles a method. If the method was annotated with HMO, the method is optimized at the level indicated. If the method was annotated with HCI, the appropriate call sites are inlined. All unannotated methods are Baseline compiled. Methods that have been optimized at the highest level and for which call site inlining is guided by HCI are not instrumented for consideration for further optimization by the JikesRVM adaptive system. All other methods (those that are Baseline compiled or optimized at lower levels) are instrumented and monitored by the system.

			Exec'd	ADAPT
	Class	Method	Methods	HOT
Program	Count	Count	100 (10)	100 (10)
compress	12	44	32 (32)	16 (10)
db	3	34	27 (24)	7 (1)
jack	56	315	265 (265)	35 (8)
javac	176	1190	740 (713)	124 (15)
jess	151	690	412 (412)	45 (10)
mpeg	55	322	201 (200)	75 (34)
mtrt	26	180	82 (81)	59 (18)
AVG	68	396	251 (247)	52 (14)

Table 3. 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)).

4. 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 benchmarks are shown in Table 3. 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 optimiza-

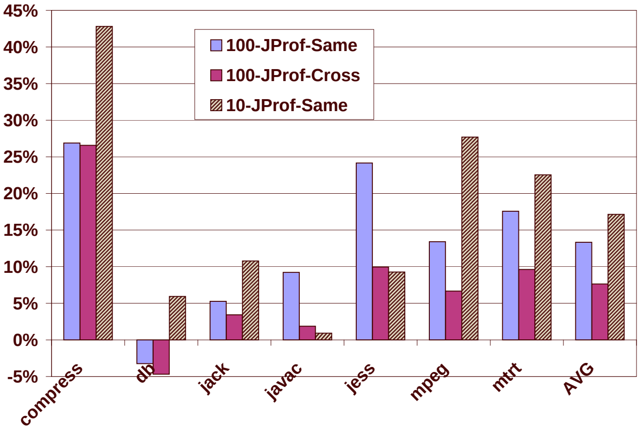


Figure 1. The percent improvement over JikesRVM (on-line profiling alone) that results when on-line and off-line profile data is combined to guide selection of methods for optimization. The first two bars show the same-input and cross-input results, respectively, for the size 100 input. The third bar shows the same-input results for the size 10 input. On average, coupling on-line and off-line profile information improves performance by 8-13% for longer running programs and 17% for short running programs.

tion). Again, data for input size 100 is shown followed by that for input size 10, in parentheses.

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.

5. Results

To empirically evaluate the efficacy of combining on-line and off-line profile information to guide optimization, we collected performance results for both Hot Method Optimization and Hot Call Site Inlining. We present these results in each of the following subsections. As part of Hot Method Optimization, we also investigated the use of both sampled and exhaustively collected profile data.

5.1 Hot Method Optimization

We first present the performance results that we achieve through the combined use of on-line and off-line profile data in JikesRVM for the Hot Method Optimization. Both types of profile data are used to select methods for optimization. All unselected methods are compiled without optimization using the JikesRVM Baseline compiler.

The first type of off-line profile collection mechanism is the sample-based approach implemented by the JikesRVM adaptive system. 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 1 since it is level most commonly selected by the JikesRVM adaptive system. We plan to consider the efficacy of annotating different levels and different individual optimizations as part of future work. All unannotated methods were instrumented for on-line measurement by the JikesRVM on-line profiling system.

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 “Adapt” column in Table 1). The graph shows percent improvement for the both the size 100 (first two bars) and size 10 inputs (third bar). For the size 100 input, we provide data for both same-input and cross-input results. We did not perform cross-input experiments for the size 10 input. 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 of the longer running programs by 8% across inputs. For the size 10 input, we improve execution time by 17%.

The benchmark for which our system causes performance degradation is db. The reason for this degradation is due to method call overhead that results from missed inlining opportunities. We address this problem with our second optimization (Hot Call Site Inlining) below. First though, we compare the efficacy of replacing sampled profile data (JProfs) with exhaustive profile data (EProfs).

5.1.1 Exhaustive Profile Data (EProfs)

As described previously, EProfs are generated using exhaustive counts of method invocations and taken loop backedges (JProfs count these events only when yield points are taken). We ordered EProf data in decreasing order so that the start of the file contained the “hottest” methods: those methods that contributed most to overall execution time.

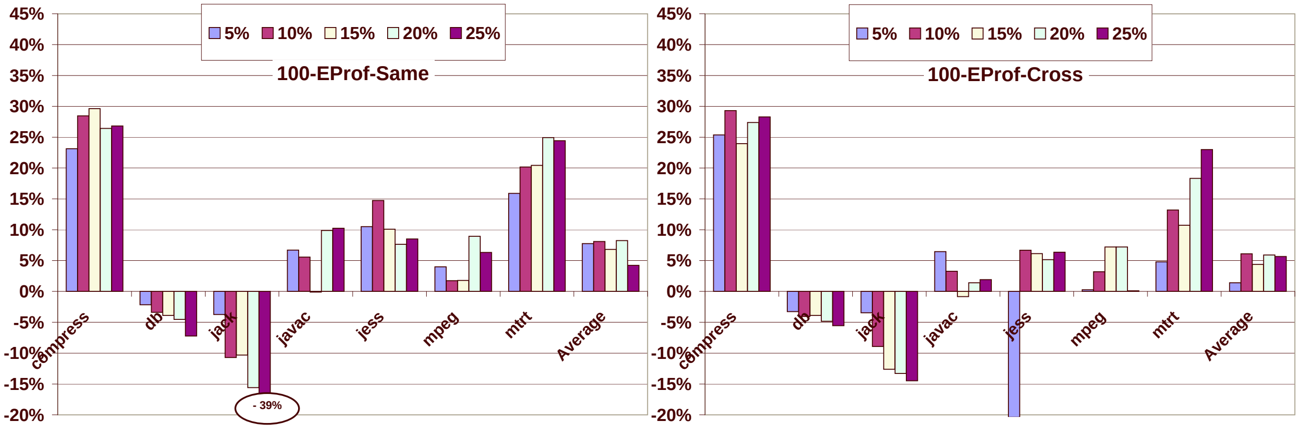
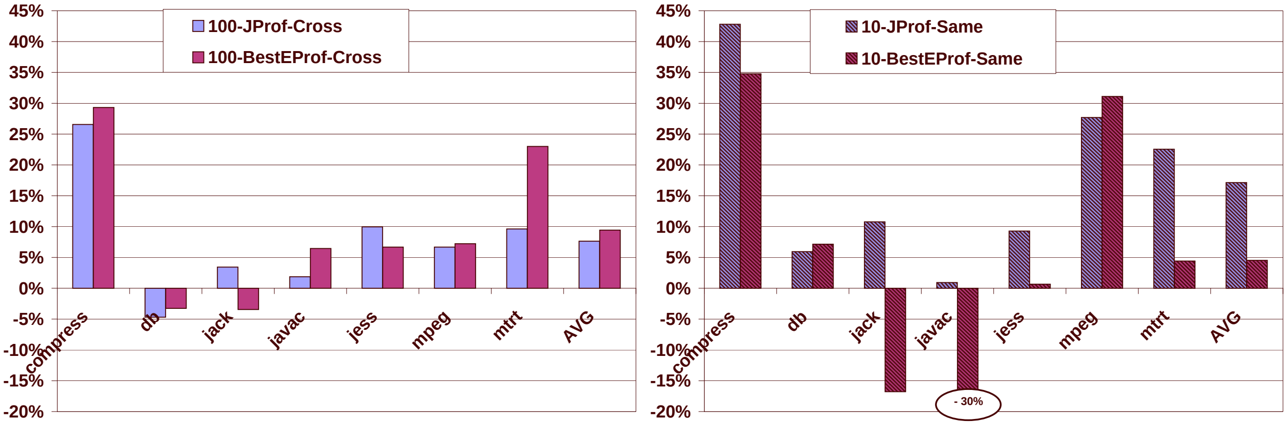


Figure 2. Percent improvement (or degradation) in performance (over JikesRVM - on-line profiling alone) enabled by coupling off-line and on-line profiles when exhaustive off-line profiles (EProfs) are used. Various percentages of hot method selection are shown. The left graph shows the same-input results and the right graph shows the cross-input results. We used the cross-input data to select percentages that perform best; we then annotate these methods in each program.



(a) (b)

Figure 3. A comparison between the use of JProfs and EProfs as off-line profile data. In the right graph, (a), the first bar is the percent improvement (over the JikesRVM on-line system alone) when JProfs (sample-based profiles) are used as the off-line profile information. This is the same data as that presented in Figure 1. The second bar is the percent improvement when the best-performing EProfs (exhaustive profiles) are used (as taken from the right graph in Figure 2). Respectively for each benchmark, the best performing cross-input EProf percentages are 10%, 5%, 5%, 5%, 10%, 20%, and 25%. Graph (b) shows the percent improvement for the size 10 input using the same input for profile and result generation. The first bar shows the JProf percent improvement and the second bar shows the EProf percent improvement (using the best-performing size 100 cross-input percentages listed above).

We next 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%, and 25%, for brevity. Higher and lower percentages did not achieve any additional improvements. 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 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 common 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. The data shown are for the size 100 inputs; the size 10 results exhibit similar behavior.

On average for the same-input results, optimizing the top 20% of methods improves performance for all benchmarks except for db and jack. For db, performance is degraded due to the overhead of method invocations. This is the same effect that is seen for the JProf results. We address this issue in the next section on Hot Call Site Inlining. Jack performance is degraded because EProfs are unable to accurately reflect the change in method "hotness" over time. Our exhaustive counts identify hot methods that are only hot for a short time. During JProf collection counts are decayed and as such, methods that are hot for only a short time are not selected for optimization. By *over optimizing*, EProfs cause degradation in the overall performance of jack.

For the benchmarks other than db and jack, the best performing percentage varies (20% is simply the best on average across benchmarks). In addition, these percentages are not the same as those across-inputs. Respectively for each benchmark, the best performing cross-input EProf percentages are 10%, 5%, 5%, 5%, 10%, 20%, and 25%. Since we are using annotation which is program and input specific, we can annotate those methods for which optimization enables the best performance across inputs. On average, using these percentages (those enabling best performance across inputs), EProfs enable 11% and 9% performance improvement over the JikesRVM on-line-only system for same-input, and cross-input experiments respectively.

An interesting characteristic in the compress benchmark data for 10% and 20% is that cross-input performance is better than same-input performance. This indicates that perfect EProf data does not accurately reflect the "hot" behavior of the program. As for jack, this is due to the lack of temporal information that is available in EProf data.

We next compare the use of JProfs with the best performing percentages of EProfs. Figure 3 (a) shows these results. The first bar (100-JProf-Cross) is the cross-input results for JProfs in Figure 1. The second bar is the the best performing bar in the right-hand graph of Figure 2. On average, across inputs, EProfs enable 9% improvement whereas JProfs enable 8% improvement. However, as we saw before, the use of EProfs cause performance degradation for jack since each is unable to account for temporal changes in method hotness.

Graph (b) in this figure shows a comparison between JProfs and EProfs for the same-input, size 10 input. The best percentages used for the EProf data are the same as those used for the results in graph (a) (which are taken from the best performing bar in Figure 2). These results indicate that for short running programs, JProfs can enable significantly better results. On average across benchmarks, JProfs enable 17% improvement and EProfs enable only 5%. Using EProfs across inputs for longer running programs offers only slight improvement over JProfs and significantly less improvement for for short-running programs. As such, we conclude that our use of sample-based profiles (generated by the JikesRVM adaptive system) is sufficient to achieve performance gains for both longer running and short running programs. As such, we consider JProfs only in the remainder of this paper.

5.2 Hot Call Site Inlining

We next consider the effect of our second optimization: Hot Call Site Inlining. For 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 guide our annotation of call sites that should be inlined.

Our results for input size 100 are shown in Figure 4 as percent improvement over the JikesRVM adaptive system (on-line-only profiling). The JProf results (first and third bars) are the same as those in Figure 1. Data resulting from the combined use of JProf and JInline profiles is shown as the second and fourth bars for each benchmark. The first pair of bars shows the same-input results and the second pair shows the cross-input results.

By using both profiles, i.e., both optimizations (Hot Method Optimization and Hot Call Site Inlining), we eliminate the db performance degradation that occurs when we use JProfs alone. In addition, our system achieves performance gains over using JProf in isolation. On average, across inputs using both JInline and JProf improves performance by 9% (15% for perfect information (same-input)). However, for jack, jess, and mpeg, the impact of combining JProfs and JInlines is less than using JProfs alone. We believe that this is due to inaccuracies in JikesRVM selection of hot call sites. We are investigating ways to improve this selection as part of future work.

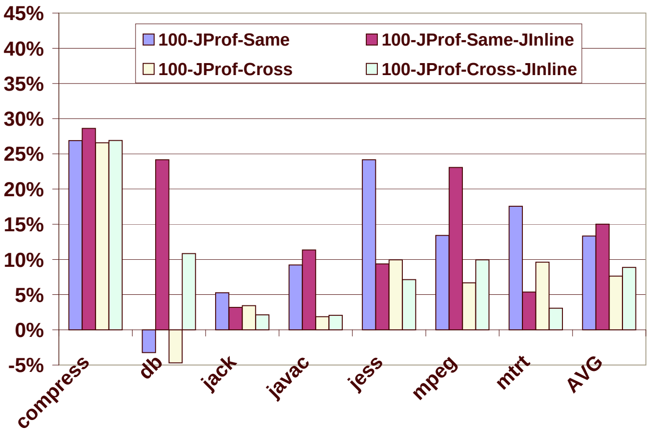


Figure 4. The effect of combining off-line and on-line profile data to guide method optimization (JProf) *and* **context-free inlining (JInline). The data is the percent improvement over the JikesRVM adaptive system for input size 100. The first and third bars are the same-input and cross-input results when JProf off-line profiles are used to guide method optimization (presented previously). The second and fourth bars show the combined effect of JProf for method optimization and JInline for inlining, respectively. For jack, jess, and mpeg, the combined use JInline and JProf profile enables less improvement than when JProf is used in isolation. We believe that this is due to JikesRVM inaccuracies in hot call site selection. We are investigating this problem further as part of future work. For db using input size 100, the performance degradation due to the use of JProfs alone is eliminated.**

6. Related Work

Various research groups have considered the use of profiles to guide dynamic Java optimization. In addition, both adaptive optimization and annotation-based systems have been extensiely studied and developed. We detail this related work in this section. The primary distinction between this prior work and that which we describe herein is that our system is the first to combine both on-line and off-line profile information dynamically to improve Java program performance.

On-line profile information has been most recently studied in the context of dynamic Java compilation [2, 6, 7]. The goal of this prior work was to reduce compilation and optimization times while enabling optimized execution speeds. These *adaptive optimization* systems insert instrumentation so that program behaviors can be measured, evaluated, and used to guide optimization decisions. In [2] and [3], the authors present an efficient sampling-based approach to on-line measurement to reduce the cost of on-line measurement. For our work, we extend this low-cost, sampling-based, on-line measurement to also consider off-line profile data to further reduce the overhead of on-line measurement.

Off-line, profile-based, optimization techniques have also been extensively researched. More recently, off-line profile information has been used to improve Java program performance in a variety of ways. For example, in prior work we used off-line program characteristics to split, prefetch, and re-order class files to reduce transfer delay [10, 11]. More recently, off-line profile information has been communicated as part of class files via annotation to reduce dynamic compilation overhead [9]. Annotation has also been used to communicate analysis information that is collected off-line to enable optimized execution times [14, 8, 4]. Our system combines the use of annotation with adaptive optimization to enable optimized execution times with very little on-line overhead.

In summary, our work is complementary to research in both adaptive optimization and annotation-based systems. Since we combine the use of both techniques in our system, any advances in these areas that improve on-line and off-line profile collection, accuracy, and use, can be immediately exploited by our system to enable further improvements in dynamic program performance.

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 disadvantages: adaptation introduces online overhead for instrumentation, measurement, and decision making, and annotations based on off-line profile information can be inaccurate. To combat these limitations and to achieve the benefits from the use of both on-line and off-line profile data, our system incorporates both to guide optimization.

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% on average for the programs studied.

8. Acknowledgments

We would like to thank the anonymous reviewers for providing extensive and useful comments on this paper.

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. 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)*, Oct. 2000.

[3] M. Arnold and B. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[4] 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.

[5] 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.

[6] 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.

[7] S. M. 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] C. Krintz, B. Calder, and U. Hölzle. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 276–291, Nov. 1999.

[11] C. Krintz, B. Calder, H. Lee, and B. Zorn. Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–169, Oct. 1998.

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

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

[14] 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`.

[15] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[16] 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.