An Empirical Study of Selective Optimization

Matthew Arnold
Rutgers University
marnold@cs.rutgers.edu

Michael Hind
IBM Watson Research Center
hind@watson.ibm.com

Barbara G. Ryder
Rutgers University
ryder@cs.rutgers.edu

March 17, 2000

Abstract

This paper describes an empirical study of the SPECjvm98 benchmarks, using the Jalapeño virtual machine. The study employs two compilers, a nonoptimizing compiler that is initially used to compile all application methods, and an optimizing compiler that is selectively used to recompile a parameterized set of hot methods based on past profiling. We view this study as a step in examining the feasibility of adaptive optimization in this environment.

The results show promise for adaptive optimization. In particular, they show that the combined time (execution and compilation) of selective opt-compilation can be less than the execution time of no opt-compilation and the combined time of full opt-compilation. The results also show that the combined time of selective opt-compilation can be competitive with static compilation (full opt-compilation not counting compilation time) for the SPECjvm98 benchmarks with input size 100.

1 Introduction

One technique for increasing the efficiency of Java applications is to compile the application to native code, thereby gaining efficiency over an interpretation execution environment. To satisfy the semantics of Java for dynamic class loading, most compilation-based systems perform compilation at runtime. However, any gains in application execution performance achieved by such JIT ("Just In Time") systems must overcome the cost of application compilation.

An alternative strategy is to *selectively compile* the methods of an application that dominate the execution time with the goal of incurring the cost of compilation for only those methods where it will be beneficial. A variant of this approach, most useful for long-running applications, such as those running on servers, is to replace the interpreter/compiler strategy with two compilers: a quick nonoptimizing compiler and an optimizing compiler. Using this approach, all methods are initially compiled by the nonoptimizing compiler and only selective methods are compiled by the optimizing compiler.

In this work we evaluate the effectiveness of such a strategy in the context of the Jalapeño Java virtual machine (JVM) [4, 10, 3, 2]. Specifically, we

- study how a past profiling run of an application can be used to understand which methods to *opt-compile*, i.e., compile with the optimizing compiler,
- define metrics that characterize how selective opt-compilation performs and report their values,
- investigate different strategies for choosing methods to opt-compile based on a past profiling run.

This work will help estimate the performance of *adaptive optimization* [18], when the selection of such methods is based on the *current* performance characteristics of the application, i.e., it does not solely depend on previous profiling runs. The results show that selective opt-compilation is a cost effective strategy; the execution time improvement obtained by opt-compiling selected methods outweighs the compile-time overhead. The results also show that in most benchmarks there is a significant range of methods that can be selected to achieve close to optimal performance. This suggests that performance benefits may be achievable with an adaptive system; such functionality is being added to Jalapeño. The rest of this paper elaborates on these points and is organized as follows. Section 2 describes the background for this study. Section 3 presents the study and discusses the results. Section 4 describes related work and Section 5 concludes.

2 Background

This experimental results used in this study were obtained using the Jalapeño JVM being built at IBM Research [4, 3, 2]. Jalapeño contains three runtime compilers, a nonoptimizing *baseline* compiler, a *quick* compiler that performs register allocation, and an *optimizing* compiler [10] discussed below. A compile-only strategy that combines executables produced by the compilers is easier than mixing compiled execution with interpreted execution. Except for a small amount of code that manages operating system services, Jalapeño is completely written in Java [3]. In addition to eliminating language barriers between the application and the JVM, writing a JVM in Java also enables optimization techniques to be applied to the JVM itself, such as inlining JVM code directly into the application [5] and adapting the JVM to the application using recompilation.

For the purposes of this study, we refer to the baseline compiler as the nonoptimizing compiler.[1] Both the nonoptimizing and optimizing compilers compile one method at a time, producing native RS/6000 AIX machine code. Jalapeño is compiled using a bootstrap process [3] in which the optimizing compiler is used to compile all methods of the JVM, including both the nonoptimizing and optimizing compilers.

The version of the optimizing compiler used for this study performs the following optimizations:

- bytecode to IR optimizations, such as copy propagation, constant propagation, dead code elimination, and register renaming for local variables [28];
- flow-insensitive optimizations, such as scalar replacement of aggregates and the elimination of local common subexpression, redundant bounds checks (within an extended basic block [12]) and redundant local exception checks;
- semantic expansion transformations of standard Java library classes [29] and static inlining, including the inlining of special methods such as lock/unlock and allocation;

These optimizations are collectively grouped together into the default optimization level (level 1). More sophisticated flow-sensitive optimizations (level 2) are under development.

On average the nonoptimizing compiler is 50 times faster than the optimizing compiler used in this study and the resulting performance benefit can vary greatly. For our benchmark suite, the execution time speedup ranges from 1.92–8.66 when the application is completely compiled by the optimizing compiler.

The Jalapeño JVM is also being used as an infrastructure to experiment with a family of interchangeable memory managers [2]. Each manager consists of a concurrent object allocator and a stop-the-world, parallel, type-accurate garbage collector. This study uses the nongenerational copying memory manager.

[1] The quick compiler is not yet fully functional and thus is not included in this study.

3 Empirical Study

Section 3.1 describes the methodology used in this study. Section 3.2 presents the main results and direct conclusions. Section 3.3 examines how accurately profiling data can be used to predict method selection. Section 3.4 further discusses these findings with respect to adaptive optimization.

3.1 Methodology

The experimental results were gathered on a 333MHz IBM RS/6000 PowerPC 604e with 1048MB RAM, running AIX 4.3. The study uses the SPECjvm98 benchmark suite [14] run on the size 10 (medium) and size 100 (large) problem sizes. In the remainder of the paper we use the terms "size 10 benchmarks" and "size 100 benchmarks" to refer to these benchmarks run on the corresponding input sizes.

In each experiment all application methods are first compiled by the nonoptimizing compiler (as would be the case with adaptive compilation) to ensure that all classes will be loaded when the optimizing compiler is invoked. This avoids any dependences due to the order in which classes are loaded. After this step a selected number of application methods are compiled again by the optimizing compiler. After this opt-compilation completes, the application is run using the results of the compilation. Thus, no compilation is performed while the application is executing.

Each benchmark/input size is run a repeated number of times. On each run N methods are opt-compiled. The manner in which methods are chosen to opt-compile is determined by (past) profile information, i.e., the application is run on size 10 input using instrumentation to determine the time spent in each method. Successive runs opt-compile the previous N methods plus the hottest remaining K methods. Initially, successive runs compile the next hottest method, i.e., $K = 1$. Once 20 methods have been opt-compiled ($N = 20$), subsequent selections use a value of 10 for K to allow for the results to be obtained in a reasonable period of time. Thus, for each benchmark/input size N methods are selected for opt-compilation, where $N = (0, 1, 2, ..., 19, 20, 30, 40, ..., \text{allmethods})$.

The time for each run is measured. This time is composed of the following three quantities: the time to compile *all* methods using the nonoptimizing compiler, the time to compile the selected methods (from 0 to allmethods) using the optimizing compiler, and the time to run the application. We refer to this total time as the *combined time*. We refer to the combined time, excluding both kinds of compilation, as the *execution time*. All times reported are the minimum of 4 runs of the application.[2]

The startup time of the Jalapeño virtual machine is not counted in the timings. Optimizing JVM startup time is beyond the scope of this paper and is a separate topic of research. The goal of this study is to better understand the tradeoffs between optimization and performance; counting JVM startup time would simply add a constant factor to all times, making the interesting tradeoff less clear.

3.2 Results

The data for each application is plotted on a graph where the x-axis measures the number of (hot) methods selected for opt-compilation and the y-axis measures time in seconds. Figures 1 and 2 present the 14 graphs, one for each application and input size. Each graph contains two curves, the solid curve connects points of the form (number-of-opt-compiled-methods, combined time). The dashed curve connects points of the form (number-of-opt-compiled-methods, execution time), i.e., the dashed curve does not include any compilation time. The time to compile all methods with the nonoptimizing compiler is included in the combined time (solid curve). This time ranged from 0.06–0.25 seconds, averaging 0.12 seconds.

The first observation concerning the graphs is the execution time difference between the size 10 and size 100 inputs to the SPECjvm98 benchmarks. With all methods optimized, the execution time only (the

[2] These results do *not* follow the official SPEC reporting rules, and therefore should not be treated as official SPEC results.

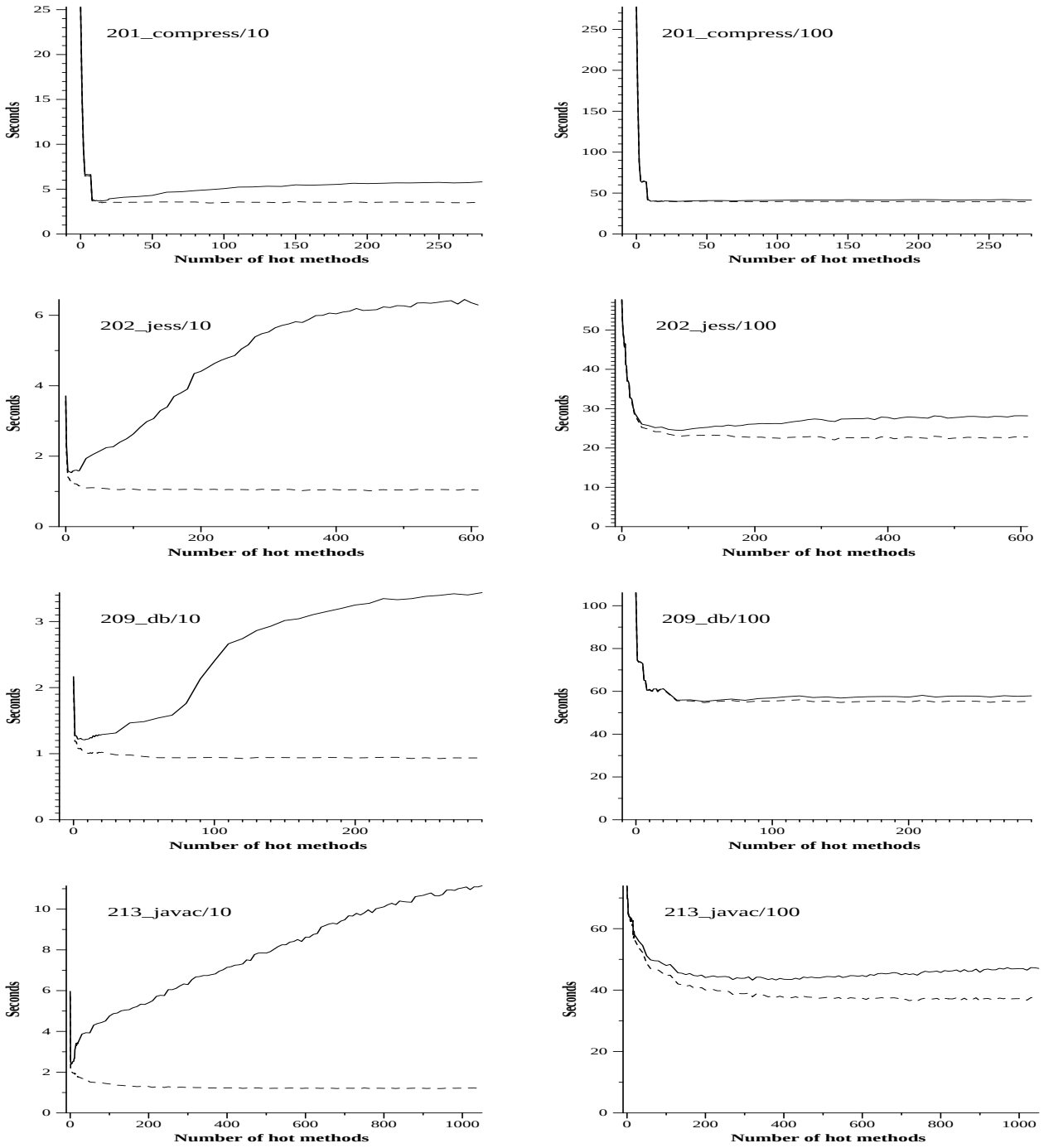

Figure 1: Graphs for `201_compress`, `202_jess`, `209_db`, and `213_javac` for input sizes 10 and 100. The x-axis is the number of methods that are opt-compiled. The y-axis is time in seconds. The dashed curve connects points for execution time. The solid curve connects points for the combined time (execution + compilation). x-axis points were recorded for $(0, 1, ..., 20, 30, 40, ..., \text{allmethods})$.

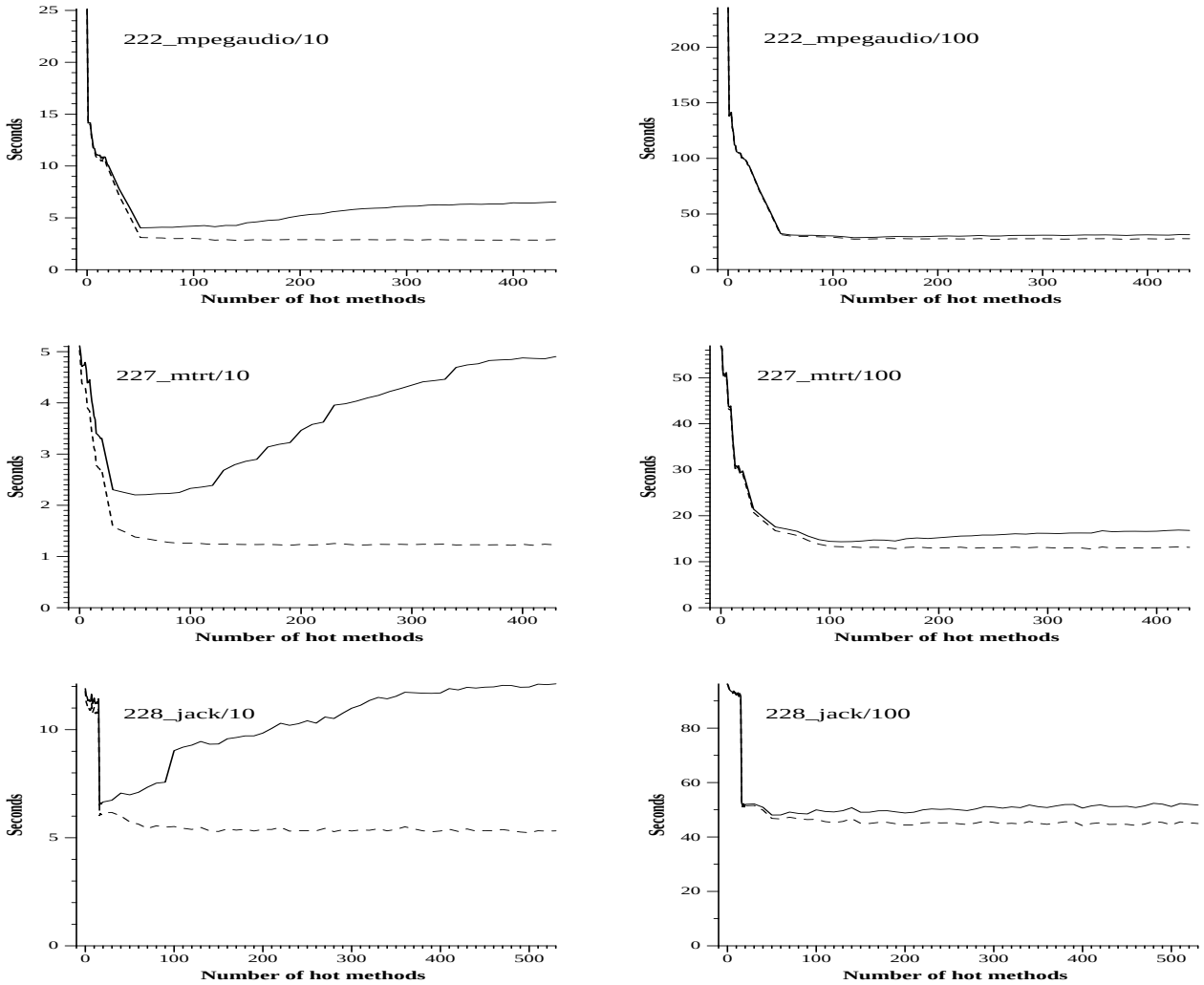

Figure 2: Graphs for `222_mpegaudio`, `227_mtrt`, and `228_jack` for input sizes 10 and 100. The x-axis is the number of methods that are opt-compiled. The y-axis is time in seconds. The dashed curve connects points for execution time. The solid curve connects points for the combined time (execution + compilation). x-axis points were recorded for $(0, 1, ..., 20, 40, 80, 120, ..., \text{allmethods})$

Benchmark/Input size	Best Characteristics: Methods for Best: Number	Best Characteristics: Methods for Best: Pct	Best Characteristics: Pct of Profile Covered	Best Performance: Best Combined Time [s]	Best Performance: Nonopt / Best	Best Performance: All (cmp+exe) / Best	Best Performance: Best / All (exe only)
201_compress/10	15	5%	99.9%	3.7	6.90	1.58	1.03
202_jess/10	8	1%	79.6%	1.5	2.42	4.11	1.47
209_db/10	8	3%	81.8%	1.2	1.79	2.84	1.29
213_javac/10	3	0%	18.1%	2.4	2.50	4.67	1.96
222_mpegaudio/10	50	11%	98.4%	4.0	6.23	1.62	1.39
227_mtrt/10	50	12%	98.5%	2.2	2.32	2.23	1.80
228_jack/10	16	3%	71.1%	6.5	1.83	1.86	1.22
Average	21.4	5%	78.2%	3.1	3.43	2.70	1.45
201_compress/100	19	7%	99.9%	39.8	6.97	1.05	1.01
202_jess/100	90	15%	99.0%	24.5	2.36	1.15	1.07
209_db/100	50	17%	98.9%	55.3	1.92	1.04	1.00
213_javac/100	320	30%	97.8%	43.2	1.71	1.09	1.16
222_mpegaudio/100	120	27%	99.8%	28.7	8.22	1.09	1.03
227_mtrt/100	110	26%	99.9%	14.3	3.97	1.17	1.09
228_jack/100	50	9%	92.3%	48.1	2.00	1.08	1.07
Average	108.4	19%	98.2%	36.3	3.88	1.10	1.06

Table 1: Effectiveness of selective opt-compilation. "All (exe only)" = execution time only, when all methods are opt-compiled. "All (cmp+exe)" = combined time (compilation and execution), when all methods are opt-compiled.

rightmost point on the dotted curve) is in the range of 0.9–5.3 seconds for the size 10 benchmarks, and 13.1–55.4 seconds for size 100. Thus it is not surprising that compilation time is a much larger percentage of combined time for size 10 inputs than for size 100 inputs, because the same methods are compiled in each case, i.e., the amount of time spent compiling is the same for a given value of N.

Another trend across the benchmarks is a steep initial drop in the curves, reaching a minimum rather quickly. After the combined time (solid curve) reaches a minimum, most of the size 10 graphs rise again, while the size 100 graphs remain relatively flat.

Table 1 elaborates on the graphs from Figures 1 and 2. In particular, it describes characteristics about the best combined time ("Best") using selective opt-compilation, i.e., the lowest point on the solid curves, and compares this time to the time of other interesting points.

The first column of Table 1 lists the benchmark name and input size. The next set of three columns reports the characteristics used to achieve the best combined time (Best) using selective opt-compilation. This could be no methods opt-compiled, all methods opt-compiled, or some methods opt-compiled in order of hotness. The column marked "Number" reports the number of (hot) methods opt-compiled. The column marked "Pct" gives the percentage of all methods in the application that are opt-compiled. The column marked "Pct of Profile Covered" gives the percentage of profile execution time that is covered by the selected methods.

The number of methods opt-compiled for Best varies greatly across the benchmarks, ranging from 3 to 320 methods (0–30% of all methods), demonstrating that the benchmarks have rather different execution patterns regarding method hotness. The percent of the profile covered by the selected methods, however, was fairly consistent among the size 100 benchmarks, averaging 98.2%.

The profile for `213_javac` has a fairly even distribution among its methods. Thus, for the size 10 input, Best includes only a small number of methods because additional opt-compilation is not beneficial on such

a short-running program. However, with the largest budget afforded by the longer-running size 100 input, Best opt-compiles the largest set of methods, 320 (30%). Despite this large number, it is the second smallest of the size 100 inputs regarding percentage of profile covered. Thus, `213_java` may be the most challenging benchmark in the SPECjvm98 suite for adaptive compilation.

The last four columns of Table 1 report the performance of the Best method selection. The first column in this set reports the combined time (execution and compilation) for the methods selected. The next three columns compare this time to other interesting values. The first of these three ("Nonopt / Best") compares the Best time to the combined time when no methods are opt-compiled. This quantity represents the performance improvement gained by selectively using the optimizing compiler (higher numbers indicate better performance for Best). The speedups range from 1.71 to 8.22 (averaging 3.66) and demonstrate that runtime optimization can significantly increase performance, even for short running and medium length programs.

The second to last column compares the Best time to the combined time (comp + exe) when all methods are opt-compiled. The resulting speedups represent the performance gains that can be achieved by avoiding optimization of infrequently executed methods (again, higher numbers indicate better performance of Best). The numbers show that selective optimization is important for the short running programs, averaging 2.70 speedup over optimizing all methods, and is also helpful for the size 100 inputs, averaging speedup of 1.10 (10%). The importance of selective optimization will only increase as more sophisticated optimizations, requiring more compile time, are added to the optimizing compiler.

The last column of Table 1 compares the Best combined (compilation and execution) time to the *execution time only* when all methods are opt-compiled. This ratio essentially compares Best against a system in which compile time is free (as is true for the static compilation model). The numbers are expressed as the additional overhead incurred by Best, so smaller numbers indicate better performance (less overhead) for Best. Not surprisingly, it is difficult for a selective opt-compilation system to match the performance of a static compiler for size 10 benchmarks because they only run for a few seconds. The size 100 benchmarks, however, offered rather encouraging results with an average slowdown of 1.06 (6%).

3.2.1 Sweet Spot

This study uses offline profiling information to determine which methods to opt-compile. An online profiling technique can benefit from more accurately capturing current information about an application. However, online profiling may not provide the same accuracy as an offline technique because it must be more efficient (it runs along side the real application run) and it will not have complete application information. Thus, one may wonder how closely a technique must estimate the set of actual hot methods that achieve Best in order to obtain reasonable performance. To address this question, we record the number of collections in order of method hotness that are within N percent of the Best combined time. We refer to these collections as being in the *sweet spot* for this benchmark. Table 1 provides this information for the following values of N: 1, 5, 10, 25, and 50. The value is expressed as the percentage of all collections within N% of the Best combined time.

This table quantifies properties that are visually present in the graphs of Figures 1 and 2. For example, the graph of `213_javac` size 10 in Figure 2 shows few points as low as the Best value at point (3, 2.39). This property is conveyed in Table 2 where the corresponding row (`213_javac/10`) shows that only 1% of the x-axis data points are within 50% of the Best combined time. This is in contrast to the graph of `209_db/100`, where many points seem to come close to the performance of Best. Table 2 shows that 90% of the selections are within 5% of Best performance. Although the size 100 inputs, in general, were more forgiving than the size 10 inputs, their 5% sweet spots did vary considerable, 20%–90%.

Benchmark/Input size	Sweet Spot Percentage				
	1%	5%	10%	25%	50%
201_compress/10	3%	3%	4%	14%	57%
202_jess/10	2%	2%	2%	2%	10%
209_db/10	3%	3%	7%	14%	24%
213_javac/10	1%	1%	1%	1%	1%
222_mpegaudio/10	5%	6%	23%	34%	57%
227_mtrt/10	7%	16%	23%	26%	40%
228_jack/10	2%	4%	9%	15%	34%
Average	3.3%	5%	9.9%	15.1%	31.9%
201_compress/100	4%	79%	96%	96%	96%
202_jess/100	3%	20%	44%	98%	98%
209_db/100	7%	90%	93%	97%	97%
213_javac/100	8%	53%	90%	96%	99%
222_mpegaudio/100	5%	25%	89%	91%	91%
227_mtrt/100	7%	23%	37%	91%	95%
228_jack/100	6%	43%	98%	98%	98%
Average	5.7%	47.6%	78.1%	95.3%	96.3%

Table 2: Percentage of x-axis data points that are within $N\%$ of Best combined time for $N = 1, 5, 10, 25, 50$.

3.3 Using Profiling as a Predictor for Method Selection

Section 3.2 determined the Best set of methods to opt-compile by observing the timings of many different executions. In an adaptive compilation system, this decision will be made by observing the profiling information and *predicting* which methods would be the best to optimize. This section investigates the effectiveness of some simple prediction strategies to determine if there exists a single strategy, for all benchmarks, that can approach the results of Best. The following predictor strategies are considered:

Num methods: Opt-compile the N hottest methods,

Percent of methods: Opt-compile $N\%$ of all application methods (in hotness order),

Time in method: Opt-compile any method that is executed for more than N milliseconds,

Relative time in method: Opt-compile any method that is executed for more than $N\%$ of the time of the hottest method,

Percent profile coverage: Opt-compile enough hot methods (in hotness order) to cover $N\%$ of the profiled execution.

For each prediction strategy a wide range of values for N were explored, with the goal of finding a value of N which performs well for all of the benchmarks of similar execution time. The value of N that resulted in the lowest performance degradation from Best, averaged over all benchmarks of that size, was chosen. The two tables in Figure 3 summarize the results. The two tables are grouped according to input size; we have found that approximate knowledge of the execution time (size 10 vs. size 100) can improve the prediction strategy. The first column of these tables lists the benchmark names and input sizes. The next five columns list the prediction strategies as described above. The line below the column headings gives the value of N that resulted in the lowest performance degradation above Best. Given this value of N, the

Benchmark/Input size	Combined Time Degradation as a Percentage of Best				
	Num Methods [50]	Percent of Methods [11%]	Time in Method [1250ms]	Relative Time in Method [0.25%]	Profile Coverage [98.29%]
201_compress/10	17.3	11.8	4.0	4.0	4.9
202_jess/10	40.0	47.6	15.1	47.2	50.6
209_db/10	22.8	12.4	6.8	20.0	21.9
213_javac/10	64.4	104.7	81.1	163.1	182.4
222_mpegaudio/10	0.0	4.8	14.4	0.0	0.0
227_mtrt/10	0.0	0.7	1.3	3.6	0.4
228_jack/10	7.5	8.9	9.3	16.4	36.7
Average	21.7	27.3	18.9	36.3	42.4

Benchmark/Input size	Combined Time Degradation as a Percentage of Best				
	Num Methods [130]	Percent of Methods [28%]	Time in Method [40ms]	Relative Time in Method [0.05%]	Profile Coverage [99.85%]
201_compress/100	4.3	2.9	0.2	4.6	4.6
202_jess/100	3.3	4.7	3.0	1.7	4.7
209_db/100	3.3	1.6	1.2	1.4	1.5
213_javac/100	5.5	1.4	0.1	1.9	4.2
222_mpegaudio/100	0.5	0.2	1.1	7.5	0.1
227_mtrt/100	1.2	0.2	0.5	16.2	1.9
228_jack/100	3.3	2.8	2.7	2.2	2.1
Average	3.1	2.0	1.2	5.1	2.7

Figure 3: Summary of the effectiveness of predictor strategies

percentage degradation above Best is listed for each benchmark and a selection policy. For example, using an 11% selection strategy the combined time for `201_compress/10` degrades 12% from Best. The average degradation percent is listed in the last line.

"Time in method" seems to be the best of the five prediction strategies, resulting in an average degradation from Best of 1.2% for size 100, and 18.9% for size 10. In general, all predictors, except "Relative Time in Method," worked reasonably well for size 100 (1.2–3.1% average degradation), and not very well for size 10 (18.9–42.4% average degradation). This is most likely not because the location of Best is inherently more difficult to predict for size 10, but because the sweet spot for size 100 is much more forgiving. For example, even though the numbers in columns 2 and 3 of Table 1 vary significantly, the predictors "Num methods" and "Percent of Methods" worked reasonably well for size 100, resulting in 2.0% and 3.1% average degradation, respectively.

3.4 Discussion

The focus of this study is the selective use of the optimizing compiler as a runtime compiler. Offline profiling is used to determine *what* to optimize (to save compile time), but not *how* to optimize. Dynamic optimizations that take advantage of up-to-date profiling information to determine how to optimize (such as specializing for common data or control values) could potentially allow adaptive compilers to outperform static compilers.

In fact, Kistler [21] concluded that his results show that runtime optimization is often better than what can be achieved at compile time (with or without profiling data).

Unlike in this study, an adaptive optimization system is not required to make all of its optimization decisions at one time. Thus the predictors presented in Section 3.3 are not the only selection options available to an adaptive compiler. An intelligent adaptive system could begin by optimizing only a small part of the application and continue until it is no longer improving performance.

The SPECjvm98 inputs are categorized as small, medium (size 10), and large (size 100), with the longest running "large" application running for less than one minute in our environment. It appears that adaptive compilation may be most effective on longer running applications, where the cost of compilation time is relatively lessened and the possibility of more indicative profiling data is greater. Increasing the compilation budget can lead to more sophisticated analyses than were used in this study.

4 Related Work

Radhakrishnan et al. [26] used the Kaffe Virtual Machine to establish the maximum performance improvement possible by interpreting, rather than compiling, cold methods. Their experiments show that between 10–15% of execution time can be saved by interpreting cold methods.

Kistler [21] presents a continuous program optimization architecture for Oberon that allows "up-to-the-minute" profile information to be used in program reoptimization. Kistler concludes that continuous optimization can produce better code than can be achieved with offline compilation, regardless of whether profiling information is used in the latter. Optimizations were evaluated by computing *ideal* performance speedup, which does not include profile or optimization time. This speedup is used to compute the *break even* point: the length of time the application would have to execute to compensate for the time spent profiling and optimizing.

Hölzle and Unger [18] describe the SELF-93 system, an adaptive optimization system for SELF. The goal of the system is avoid long pauses in interactive applications by optimizing only the performance-critical parts of the application. Method counters with an exponential decay mechanism are used to determine candidates for optimization. On average they found an improvement of 160% over a system in which all methods are optimized. They concluded that their experiments showed that determining the selection mechanism ("what" to optimize) was less important than determining the trigger mechanism ("when" to optimize).

Bala et al. [7] describe Dynamo, a transparent dynamic optimizer that performs optimizations at runtime on a native binary. Dynamo initially interprets the program, keeping counters to identify sections of code called hot *traces*. These sections are then optimized and written as an executable. The authors report an average speedup of 7% and an average overhead of 1.4%.

Burger and Dybvig [8, 9] explore profile-driven dynamic-recompilation in Scheme, with an implementation of a basic block reordering optimization. They report a 6% reduction in runtime and an average recompile time of 12% of their base compile time.

Hansen [17] describes an adaptive FORTRAN system that makes automatic optimization decisions. When a basic block counter reaches a threshold, the basic block is reoptimized and moved to the next *optimization state*, where more aggressive optimizations are performed.

The HotSpot JVM [20] and the IBM Java Just-in-Time compiler (version 3.0) [27] are adaptive systems for Java. Both systems initially interpret an application and later compile performance-critical code. The IBM JIT uses method invocation counters augmented to accommodate loops in methods to trigger compilation. Details of the HotSpot compilation system are not provided.

Another area of research considers performing runtime optimizations that exploit invariant runtime values. Because such values are not statically determinable, these systems provide optimization opportunities not available with static compilation. Some systems include *DyC* [6, **?**, 16] and *Tempo* [24] (based on C) and

Fabius [23] (based on ML), as well as Consel and Noel's work [13] which takes a partial evaluation approach. The *tcc* system [25] provides a mechanism to specify and compose arbitrary expressions and statements at runtime. The main disadvantage of these techniques is that they rely on programmer directives to identify the regions of code to be optimized.

There are also nonadaptive systems that perform compilation/optimization at runtime to avoid the cost of interpretation. This includes early work such as the Smalltalk-80 [15] and Self-91 [11] systems, as well many of today's JIT Java compilers [1, 22, 30]. The version of the Jalapeño system used in this work is also in this category, although work is currently underway to make it an adaptive system.

There are also fully automated profiling systems that use transparent profiling to improve performance of future executions. Such systems include Digital FX!32 [19] and *Morph* [31]. However, these systems do not perform compilation/optimization at runtime.

5 Conclusions

This paper described an empirical study of selective opt-compilation using the Jalapeño JVM and the SPECjvm98 benchmarks. The main conclusions are

- The Best number of methods to opt-compile varies greatly across the benchmarks, ranging from 3 to 320 methods (0–30% of all methods). The percent of the profile covered by the selected methods is fairly consistent among the size 100 benchmarks, averaging 98.2%.
- The Best selective opt-compilation strategy (counting compilation time) outperformed a no opt-compilation strategy by a speedup of up to 8.22, (average 3.66), demonstrating that runtime optimization can significantly increase performance.
- The Best selective opt-compilation strategy outperformed an all opt-compilation strategy, averaging a 2.70 speedup for size 10 inputs and a speedup of 1.10 for size 100 inputs.
- The Best selective opt-compilation strategy (counting compilation time) is competitive with the all opt-compilation strategy when compilation time is excluded, averaging a slowdown of only 1.06 (6%) on the size 100 benchmarks.
- The "Time in method" prediction strategy seemed to be the best of the five simple prediction strategies studies, resulting in an average degradation from Best of 1.24% for size 100, and 18.86% for size 10.

Acknowledgments

We thank the Jalapeño team members [2] for their work in developing the system used to conduct this study. In particular, David Grove, Vivek Sarkar, and Peter Sweeney provided useful suggestions and feedback on this work. We also thank Michael Burke for his support and Stephen Fink and Laureen Treacy for comments on a earlier draft of this work.

References

[1] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998.

[2] 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 Jounral*, 39(1), 2000.

[3] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[4] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a compiler-supported Java virtual machine for servers. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 36–46, May 1999. INRIA Technical Report #0228.

[5] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter Sweeney. A comparative study of static and dynamic heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[6] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, 21–24 May 1996. *SIGPLAN Notices* 31(5), May 1996.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical report hpl-1999-78, Hewlett Packard Laboratories Cambridge, June 1999.

[8] Robert G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme.* PhD thesis, Indiana University, 1997.

[9] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *ICCL'98, the IEEE Computer Society International Conference on Computer Languages*, May 1998.

[10] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[11] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, November 1991. *SIGPLAN Notices* 26(11).

[12] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31, September 1999.

[13] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 145–156, New York, NY, USA, 1996. ACM Press. ACM order number: 549960.

[14] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[15] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, January 1984.

[16] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, 1999.

[17] Gilbert J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs.* PhD thesis, Carnegie-Mellon University, 1974.

[18] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[19] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining emulation and binary transslation. *Digital Technical Journal*, 9(1):3–12, January 1997.

[20] The Java Hotspot performance engine architecture. White paper available at http://java.sun.com/products/hotspot/whitepaper.html, April 1999.

[21] Thomas P. Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, 1999.

[22] Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, October 1998.

[23] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):1–5, September 1998. Article 23.

[24] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 281–292, 1999.

[25] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 109–121, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.

[26] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture (HPCA-6)*, pages 387–398, Toulouse, France, January 2000.

[27] Toshio Suganama, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Jounral*, 39(1), 2000.

[28] John Whaley. Dynamic optimization through the use of automatic runtime specialization. M.eng., Massachussetts Institute of Technology, May 1999.

[29] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, San Fransisco, CA, June 1999.

[30] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[31] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 15–26, New York, October 5–8 1997. ACM Press.