Workshop Version

In 13th International Workshop on Languages and Compilers for Parallel Computing

An Empirical Study of Selective Optimization

Matthew Arnold[1,2], Michael Hind[2], and Barbara G. Ryder[1]

[1]Rutgers University
110 Frelinghuysen Road, Piscataway, NJ 08854, USA
{marnold,ryder}@cs.rutgers.edu
[2]IBM Thomas J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598, USA
hind@watson.ibm.com

Abstract. This paper describes an empirical study of selective optimization using the Jalapeño Java virtual machine. The goal of the study is to provide insight into the design and implementation of an adaptive system by investigating the performance potential of selective optimization and identifying the classes of applications for which this performance can be expected. Two types of offline profiling information are used to guide selective optimization, and several strategies for selecting the methods to optimize are compared.
The results show that selective optimization can offer substantial improvement over an optimize-all-methods strategy for short-running applications, and for longer-running applications there is a significant range of methods that can be selectively optimized to achieve close to optimal performance. The results also show that a coarse-grained sampling system can provide enough accuracy to successfully guide selective optimization.

1 Introduction

One technique for increasing the efficiency of Java applications is to compile the application to native code, thereby gaining efficiency over an interpreted environment. To satisfy Java's dynamic semantics, such as 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 optimize* the methods of an application that dominate the execution time with the goal of incurring the cost of optimization for only those methods in which the performance of compiled code will be most beneficial. One example of this approach is to use two compilers: a quick nonoptimizing compiler and an optimizing compiler. With this compile-only strategy, all methods are initially compiled by the nonoptimizing compiler and only selective methods are optimized.

The goal of this work is to empirically evaluate the limitations of selective optimization. Although systems that selectively optimize methods during execution do exist [20, 22, 29, 7, 14, 5] the focus of such work is typically overall system performance, which is affected by many characteristics of the particular system being studied. These characteristics include 1) when a recompilation decision is made, 2) the overhead and online nature of the profiling information, 3) the sophistication and overhead of the recompilation decision-making component, and 4) the efficiency and performance increase obtained by using the optimizing compiler. This study holds these factors constant to provide a comprehensive evaluation of the effectiveness of selective optimization. Namely, 1) all optimization occurs before execution; 2) offline profiling is used; 3) optimization decisions are made before execution is begun; and 4) the Jalapeño optimizing compiler [11] is used.

In this work we

- establish the performance gains possible for selective optimization and identify classes of applications for which these gains can be expected;
- define metrics that characterize the performance of selective optimization and report their values;
- compare the performance of selective optimization when driven by two different profiling techniques, time-based and sample-based; and
- investigate different strategies for choosing methods to optimize.

The results provide useful insight for the design and implementation of an adaptive system. We have observed that selective optimization can offer substantial improvement over an all-opt (optimize all executing methods) strategy for our short-running benchmark suite, and for the longer-running suite there is a significant range of methods that can be selected to achieve close to optimal performance. The results also show that a coarse-grained sampling system can provide the accuracy necessary to successfully guide selective optimization, suggesting that the performance results may be obtainable with an adaptive system. Indeed, the results from this work were used during the design of the Jalapeño adaptive system [5].

The rest of this paper elaborates on these points and is organized as follows. Sections 2 and 3 describe the background and methodology used in this study. Section 4 presents the main results and direct conclusions. Section 5 investigates the performance of several automatic method selection strategies. Section 6 describes related work. Section 7 draws conclusions and discusses these findings with respect to adaptive optimization.

2 Background

The results presented in this study were obtained using the Jalapeño JVM [2, 3] being developed at IBM T.J. Watson Research Center. Except for a small amount of code that manages operating system services, Jalapeño is written completely 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 and adaptively optimizing the JVM to the application.

Jalapeño currently contains two fully operational compilers, a nonoptimizing *baseline* compiler and an *optimizing* compiler [11]. The version of the optimizing compiler used for this study performs the following optimizations:

- on-the-fly optimizations during bytecode to IR translation, such as copy propagation, constant propagation, dead code elimination, and register renaming for local variables [31];
- flow-insensitive optimizations, such as scalar replacement of aggregates and the elimination of local common subexpressions, redundant bounds checks (within an extended basic block [13]) and redundant local exception checks;
- semantic expansion transformations [32] of standard Java library classes and static inlining, including the inlining lock/unlock and allocation methods.

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

Both the nonoptimizing (baseline) and optimizing compilers compile one method at a time, producing native RS/6000 AIX machine code. On average the nonoptimizing compiler is 76 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 1.22–8.17 when the application is completely compiled by the optimizing compiler.

Jalapeño provides a family of interchangeable memory managers [2] with a concurrent object allocator and a stop-the-world, parallel, type-accurate garbage collector. This study uses the non-generational copying memory manager.

3 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[1] benchmark suite [16], Jalapeño's optimizing compiler [11], the pBOB multithreaded business transaction benchmark [8], and the Volano benchmark [30], which is a multithreaded server application that simulates chat rooms. The benchmarks range in cumulative class file sizes from 10,156 (209_db) to 1,516,932 (opt-cmp) bytes.

The performance runs are grouped into two categories: *short running* and *longer running*. The short-running category includes the SPECjvm98 benchmarks using the size 10 (medium) inputs and the Jalapeño optimizing compiler compiling a small subset of itself to give a comparable runtime. The `pBOB` and `Volano` benchmarks are designed to be longer-running server benchmarks and thus they are not included in the shorter running benchmarks. The longer-running category includes the SPECjvm98 benchmarks using the size 100 (large) inputs, the Jalapeño optimizing compiler compiling a larger subset of itself, and the `pBOB` and `Volano` benchmarks. The `pBOB` and `Volano` benchmarks run for a fixed amount of time and report their performance as a throughput metric in events per second. To allow comparison, we report an equivalent metric of seconds per 750,000 transactions for `pBOB` and seconds per 140,000 messages for `Volano`.

Each experiment first compiles all application methods using the nonoptimizing compiler (as is the case in a compile-only adaptive system) to ensure that all classes will be loaded when the optimizing compiler is invoked. This avoids (in optimized code) the performance penalties associated with dynamic linking, i.e., calls to methods of classes that have not been loaded, and also allows the called method to be a candidate for inlining. Next, a selected number of application methods are compiled again by the optimizing compiler. Finally, the application is executed without any further compilation being performed. Thus, the application's execution time contains no compilation overhead.

Each benchmark is executed a repeated number of times, where the number of methods optimized, N, is increased. These methods are chosen in order of hotness, based on a past profile of the unoptimized application with the same input. Thus, the number of methods that can be optimized is bound by the number of profiled methods. For each benchmark we chose the following values of N: 0, 1, 2, ..., 19, 20, 30, 40, ..., # profiled methods. We use the term "all methods" to refer to all methods present in the profile.

For each value of N, three quantities are measured: nonoptimizing compilation time (of all methods), optimization time (of N methods), and execution time. We refer to the total of these three times as the *combined time.* All times reported are the minimum of 4 runs of the application for the given value of N. The startup time of the Jalapeño virtual machine is not counted in the timings because it would simply add a constant factor to all times. Optimizing startup time of a JVM is a separate research topic.

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

3.1 Profiling

Two types of profiling information are used to determine the order in which methods are selected for optimization: *time-based* and *sample-based.* The time-based profile records time spent in each method, and is collected by instrumenting all calls, returns, throws, and catches to check the current time and add it to the appropriate method. Methods are ordered based on the total time spent in the method.

The sampling-based profiling technique samples the running application approximately once every 10 milliseconds when the application reaches predefined sample points. These points are located on entry to a method and on loop back edges. Samples taken on loop back edges are attributed to the method containing the loop. Samples taken on method entries are attributed to both the calling and called method. Methods are ordered based on the total number of samples attributed to them. This sampling technique is efficient — the overhead introduced does not stand out above the noise from one run to the next — and it is being used in the initial Jalapeño adaptive system [5].

The method orderings from both profiles are likely to contain a certain degree of imprecision. Sampling every 10 milliseconds is fairly coarse-grained considering the speed of the processor; thus the sampled profile will contain some imprecision, especially for short-running applications. The time-based profile is also imprecise because the instrumentation introduces overhead, increases code size, and disrupts the instruction cache.

A major difference between two profiles is that the time-based profile is complete in the sense that all executing methods will appear in the profile. The sample-based profile is an incomplete, or partial profile, because it is possible for a method to execute but never be sampled.

4 Results

This section presents the main results of the study. Section 4.1 describes the results using the time-based profile. Section 4.2 describes the results using the sample-based profile and compares these results to those from Section 4.1.

4.1 Time-Based Profile

Each application in the two suites can be plotted on a graph where the x-axis measures the number of (hot) methods selected for optimization and the y-axis measures time in seconds. Fig. 1 shows representative graphs for the short-running and longer-running suites using the time-based profile.[2] Each graph contains two curves, a solid curve connects points of the form (number-of-optimized-methods, combined time). A dashed curve connects points of the form (number-of-optimized-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 0.06–0.63 seconds, averaging 0.19 seconds.

One observation to make is the execution time difference between the short-running and longer-running benchmark suites. With all methods optimized, the execution time only (the rightmost point on the dotted curve) is in the range of 1.1–4.8 seconds for the short-running benchmarks, and 16.6–70.6 seconds for longer-running benchmarks. Thus it is not surprising that compilation time is a larger percentage of combined time for short-running benchmarks than for the longer-running benchmarks.

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.

[2] Due to space constraints we are unable to show all graphs.

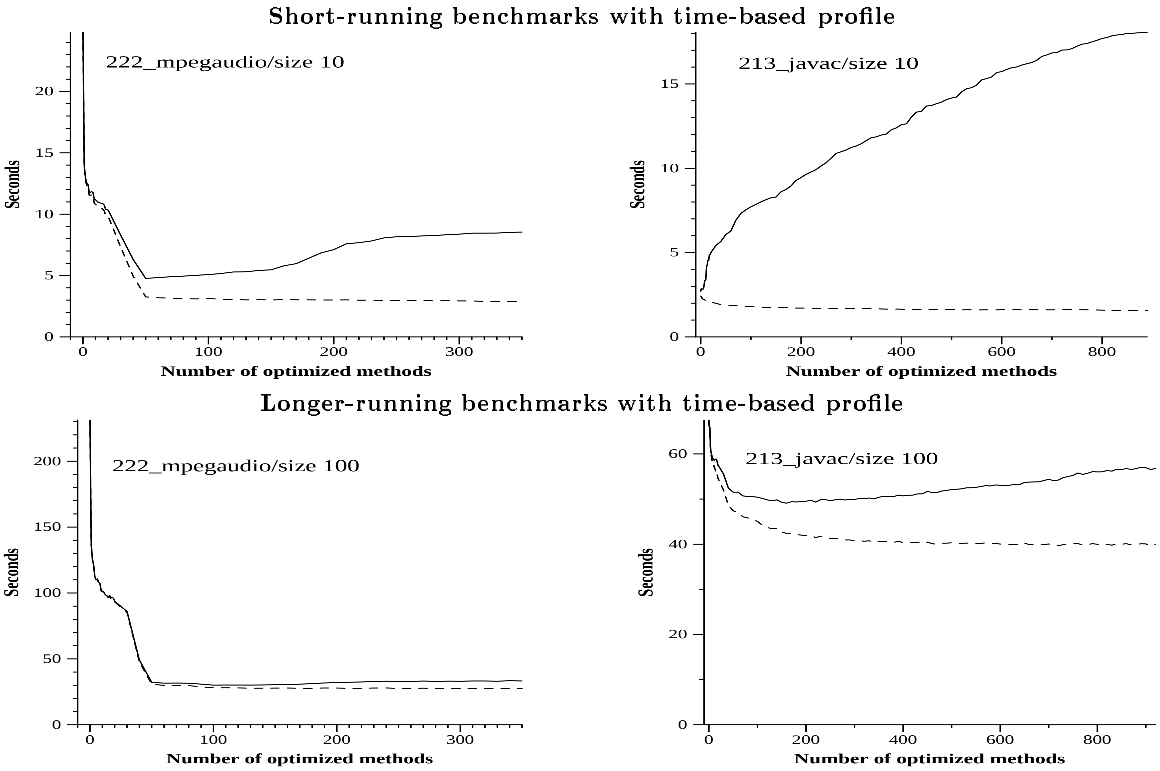

Fig. 1. Graphs for representative short-running and longer-running benchmark suites for the time-based profiles. The x-axis is the number of methods that are optimized. The y-axis is time in seconds. The dashed curve connects points for execution time. The solid curve connects points for the combined time (execution + compilation). x-axis points are recorded for $(0, 1, ..., 20, 30, 40, ..., \#$ profiled methods$)$.

Table 1. Effectiveness of selective optimization using time-based profiling for both the short and longer-running benchmarks. Recall that `pBOB` and `Volano` are not included in the short-running suite because they are designed to be server benchmarks.

	Short-running suite				Longer-running suite			
	Best Characteristics				Best Characteristics			
	Methods for Best		Pct of Profile	Best Combined	Methods for Best		Pct of Profile	Best Combined
Benchmark	Number	Pct	Covered	Time [s]	Number	Pct	Covered	Time [s]
201_compress	16	9%	99.9%	3.8	15	8%	99.9%	39.9
202_jess	7	1%	88.6%	1.7	40	8%	98.9%	31.3
209_db	4	2%	84.6%	1.5	9	5%	99.9%	70.9
213_javac	0	0%	0%	2.7	160	17%	92.9%	49.1
222_mpegaudio	50	14%	99.8%	4.8	100	29%	99.9%	30.0
227_mtrt	20	6%	86.3%	3.8	70	21%	99.2%	19.7
228_jack	3	1%	40.5%	5.8	13	3%	82.4%	44.3
opt-cmp	1	0%	11%	11.2	290	18%	95.6%	75.8
pBOB	—	—	—	—	200	33%	98.9%	54.8
Volano	—	—	—	—	13	2%	93.9%	38.6
Average	12.6	3.8%	63.8%	4.4	91.1	14.4%	96.1%	45.4

Table 1 describes characteristics of the best combined time ("Best") using selective optimization, i.e., the lowest point on the solid curves, for all benchmarks, using the time-based profile. The first column of Table 1 lists the benchmark name. The remaining columns present the results for the two categories of benchmarks, four columns for the short-running programs and four columns for the longer-running programs. Within each category the first three columns report the characteristics used to achieve the best combined time (Best) using selective optimization. This could be no methods optimized, all methods optimized, or some methods optimized in order of hotness. The column marked "Number" reports the number of (hot) methods optimized. The column marked "Pct" gives the percentage of all methods in the application that are optimized. The column marked "Pct of Profile Covered" gives the percentage of the time-based profile that is covered by the selected methods. The following column gives the Best time in seconds.

The number of methods optimized for Best varies greatly across the benchmarks, ranging from 0 to 290 methods (0–33% of all methods), demonstrating that the benchmarks have rather different execution patterns regarding method hotness. For example, `201_compress` spends 99.9% of its execution time in just 9% of its methods. This is in contrast to the more object-oriented benchmarks, such as `213_javac`, `opt-cmp` and `pBOB`, which distribute their execution time more evenly among a much larger set of methods, and thereby presenting a bigger challenge for selective optimization. For the short-running benchmarks, Best includes 0 and 1 methods for `213_javac` and `opt-cmp`, respectively, covering only 0% and 11% of the time-based profile. For the longer-running benchmarks, Best selects the largest number of methods to optimize for the three object-oriented benchmarks. Yet despite the large number of methods optimized, these benchmarks are among the four smallest regarding percentage of profile covered.

The bar charts in Figs. 2 and 3 compare the combined time (compilation + execution) performance of three interesting method selection strategies: no methods optimized (none-opt), all methods optimized (all-opt), and Best. Recall that Best, by definition, gives the best combined time, so it will always be better than or equal to the other two configurations. Below each benchmark name is the *execution time only* when all methods are optimized (corresponding to the right-most point on the dashed lines in Fig. 1), which we refer to as *all-opt-exe*. All bars are normalized to this value; thus, all three method selection strategies are shown relative to a system in which compile time is free and all methods run at peak (for this environment) performance.[3] Given Java's dynamic semantics, it does not appear that such a system, without any runtime compilation or interpretation, is possible.

For the short-running benchmarks (Fig. 2), Best averages an additional 67% slower than all-opt-exe. The all-opt scheme is the worst of the three selections for 6 of the 8 benchmarks. This in not surprising because the benchmarks do not run long enough to recover the cost of optimizing all methods. For 5 of 8 benchmarks, the improvement of Best over the other two extremes is substantial. For all benchmarks it averages 56% better than none-opt and 71% better than all-opt.

For the longer-running benchmarks (Fig. 3), Best averages only an additional 13% in combined time compared to an all-opt-exe configuration. All-opt is much closer to Best for all cases, averaging 28% worse than all-opt-exe. Because of the benchmarks longer execution time, none-opt is the worst strategy in all cases. These results suggest that although an all-opt strategy is effective for longer-running applications the optimization overhead is not insignificant.

Sweet Spot The previous performance results for Best are based on a particular ordering of the methods using an offline time-based profile. However, one may wonder how closely a technique must estimate the set of actual hot methods that achieve Best to obtain reasonable performance. To address this question, we record the number of collected timings that are within N percent of the

[3] An adaptive system can utilize feedback-directed optimizations, i.e., optimizations based on the application's current execution profile, to potentially obtain performance beyond that of all-opt-exe.

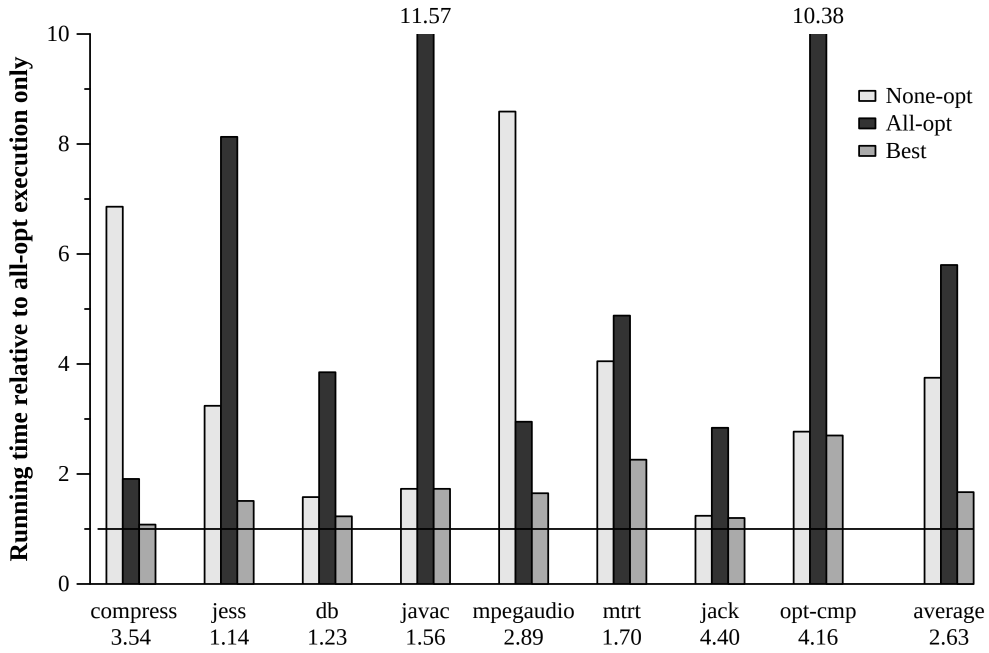

Fig. 2. Short-running suite performance comparison. The height of each bar is the combined time normalized to all-opt-exe time. The numbers under the benchmark names are the all-opt-exe time in seconds.

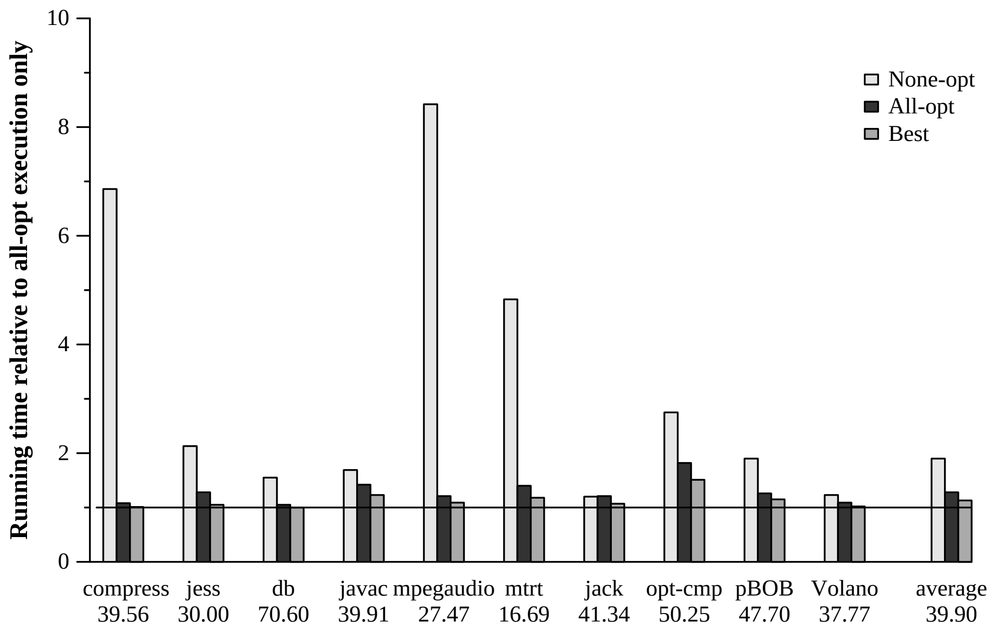

Fig. 3. Longer-running suite performance comparison. The height of each bar is the combined time normalized to all-opt-exe time. The numbers under the benchmark names are the all-opt-exe time in seconds.

Best combined time. We refer to this number as being in the *sweet spot* for this benchmark. Table 2 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. Due to space considerations we present for each sweet spot range the average and range of percentages over the benchmarks in each suite. For example, the value of 2.6% in the "1%" column states that on average, 2.6% of the collected timings of the short-running benchmarks fell within 1% of the Best time. The next row, "1–5%" states that 2.6% represents an average of values that ranged from 1–5%.

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

		Sweet Spot Percentage				
Benchmark	Metric	1%	5%	10%	25%	50%
Short running	Avg	2.6%	3.9%	6.2%	13.0%	23.0%
Short running	Range	1–5%	1–11%	1–20%	1–34%	1–50%
Longer running	Avg	7.1%	33.1%	62.2%	89.4%	92.5%
Longer runnnig	Range	1–16%	5–95%	10–95%	25–100%	50–100%

This table quantifies properties that are visually present in the graphs of Fig. 1 as well as those graphs not shown. For example, the graph of the short-running `213_javac` in Fig. 1 shows few points as low as the Best value at point (0, 2.39), i.e., the no methods optimized resulted in the best running time. In fact, for this benchmark only 1% of the x-axis data points are within 50% of the Best combined time. This is in contrast to the graph of the longer-running `222_mpegaudio`, where 90% of the selections are within 5% of Best performance. Although the longer-running benchmarks, in general, are more forgiving than the short-running benchmarks, their 5% sweet spots do vary considerable, 5%–95%.

4.2 Sample-Based Profile

As mentioned in Section 3.1, a sample-based profile was also used to examine the feasibility of selective optimization. The same experiments were repeated, except the sample-based profile was used to establish method hotness. Fig. 4 shows two example performance graphs using the sample-based profile. The curves start out similar to those from the time-based profile; however, the curves end much earlier because only a subset of the executing methods are sampled.

Table 3 shows characteristics of the sample-based profile for all benchmarks. The second column shows the percentage of all executed methods that are sampled, ranging from 6.7%–57.5% and averaging 24.6%. The remaining columns of Table 3 show characteristics about the Best from the sampled-profile (sampled-best), and compare it to Best from the time-based profile (time-best). The three columns marked "Num Methods for Best" give the number of methods that are optimized to achieve time-best and sample-best, respectively, and the number of methods optimized in both cases, i.e., the intersection of the methods of the two previous columns. For the short-running benchmarks, the methods selected for Best are similar for both profiles; however, for the longer-running benchmarks there is a fairly large variation, especially for `213_javac`, `opt-cmp`, and `pBOB`.

The last two columns of Table 3 show the running time of sample-best and the ratio of sample-best to time-best. Sample-best shows almost no degradation to time-best, and in several cases is better than time-best. As mentioned in Section 3.1, sample-best can be better than time-best because instrumentation perturbs the code. Thus, it is possible for the hotness ordering generated by the

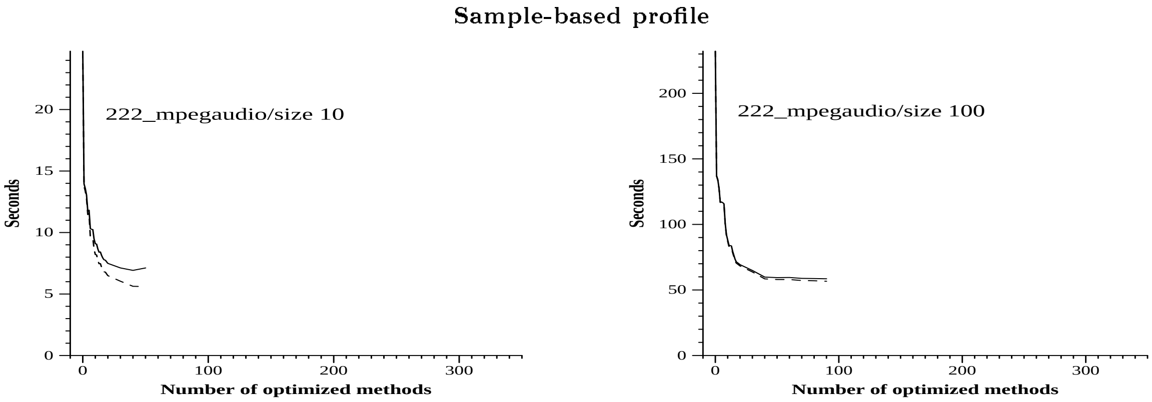

Fig. 4. Graphs for representative short-running and longer-running benchmark suites for the sample-based profiles

Table 3. Comparison of "Best" using the sample-based and time-based profile

	Benchmark	% Methods Sampled	Num Methods for Best: Time	Sample	Both	Sampled Best Time	Sampled Best / Time Best
Short-running suite	201_compress	8.5%	16	10	10	3.8	1.00
	202_jess	6.7%	7	4	4	1.7	1.00
	209_db	9.9%	4	8	3	1.6	1.02
	213_javac	13.6%	0	0	0	2.7	1.00
	222_mpegaudio	15.5%	50	40	28	4.8	1.01
	227_mtrt	26.0%	20	20	14	3.9	1.01
	228_jack	17.7%	3	5	4	5.7	0.99
	opt-cmp	29.7%	1	17	1	9.4	0.84
	Average	15.9%	12.6	15.5	8.0	4.2	0.98
Longer-running suite	201_compress	11.0%	15	11	10	39.9	1.00
	202_jess	15.3%	40	40	36	31.4	1.00
	209_db	8.8%	9	7	5	70.6	1.00
	213_javac	41.9%	160	160	122	49.3	1.01
	222_mpegaudio	27.2%	100	90	64	30.2	1.01
	227_mtrt	34.8%	70	60	50	19.2	0.97
	228_jack	33.4%	13	30	12	43.8	0.99
	opt-cmp	57.5%	290	220	203	75.2	0.98
	pBOB	46.2%	200	210	157	53.4	0.97
	Volano	13.6%	13	12	2	39.0	1.01
	Average	28.9%	91.0	88.0	67.9	45.2	0.99

sampled profile to be more accurate than the time-based profile. Furthermore, because all calls and returns are instrumented, the error is likely to be greatest for benchmarks that contain a large number of short-lived method calls.[4]

The most important observation to be made from this data is that the sample-based, incomplete profile achieves the same Best performance as the time-based profile *independent of the running time of the application.* The key is that the sample-based profile is accurate enough for the methods that matter. For the short-running applications there is not much time to optimize, so only the hottest few methods need to be detected by the sample-based profile. As the program runs longer and there is more time to optimize, the sample-based profile will have had enough time to reasonably order the needed number of methods.

5 Using Profiling as a Predictor for Method Selection

Section 4 determined the Best set of methods to optimize by observing the timings of many different executions. In an adaptive optimization 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 strategies are considered:

Num methods: Optimize the N hottest methods,

Percent of methods: Optimize $N\%$ of all profiled application methods (in hotness order),

Time/Samples in method: Optimize any method that is executed for more than N milliseconds or N samples,

Time/Samples in method relative to hottest method: Optimize any method that is executed for more than $N\%$ of the time of the hottest method,

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

For each 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 similar size, is chosen.

Table 4 summarizes the results for the short-running and longer-running suites, respectively, for both types of profiles. The first column lists the profile used and benchmark suite. The last five columns list the strategies as described above. For each pair of profile type and benchmark suite, there are three rows. The first row gives the value of N that resulted in the lowest average performance degradation above Best. Given this value of N, the second and third rows show the average percentage degradation from (worse than) Best, and the range for all benchmarks in that suite, respectively. For example, for the short-running benchmarks with the time-based profile, optimizing the 9 hottest methods resulted in an average degradation of 31% from Best for those benchmarks, with values ranging from 0.1%–71.0%.

The first observation is that Best is harder to predict for the short-running benchmarks (first and third group of rows) than for the longer-running benchmarks (second and fourth group of rows), as can be seen by comparing the rows labeled "Avg". The average degradations from Best for the short-running suite varies across the different prediction strategies from 27%–53% for the time-based profile and 6.2%–50.2% for the sample-based profile. For the longer-running benchmarks, the worst

[4] The methodology used for selecting methods to optimize does not consider characteristics of the methods such as compile time or the performance improvement of optimization. Therefore, it is possible that either profiling technique could "get lucky" if it happens to bias small methods, or methods that optimize well.

Table 4. Summary of the effectiveness of predictor strategies

Profile Used/ Benchmark Suite	Metric		Num Methods	Pct of Methods	Time/ Samples in Method	Time/Samples in Method Relative to Hottest Method	Pct Profile Coverage
Time/ Short running	Optimal Values		9	5%	38810ms	4.7%	74.3%
	Pct Decrease	Avg	31.7	49.1	27.2	41.5	53.8
		Range	0.1–71.0	1.8–116	0.0–73.5	0–130	0.2–135
Time/ Longer running	Optimal Values		120	19%	253ms	0.05%	99.7%
	Pct Decrease	Avg	3.7	2.5	3.6	3.7	3.7
		Range	0.2–6.6	0.2–11.9	1.5–17.2	0.1–6.7	0.2–7.3
Sample/ Short running	Optimal Values		10	11%	10	3.25%	70.25%
	Pct Decrease	Avg	6.2	35.0	8.5	42.2	50.2
		Range	0.0–31	0.0–148	0.0–51.4	0.9–135	0.9–147
Sample/ Longer running	Optimal Values		180	50%	9	0.10%	99.2%
	Pct Decrease	Avg	2.5	1.2	1.0	2.7	2.3
		Range	0.0–7.7	0.0–2.5	0.0–2.2	0.0–7.4	0.1–9.7

average degradation from both profiling techniques is 3.7%. This extreme difference between the short- and longer-running suites is most likely not because the location of Best is inherently more difficult to predict for short-running applications, but because the sweet spot of the longer-running applications is much more forgiving.

"Time/Samples in method" is the most effective overall strategy; its average degradation from Best is either the smallest, or close to the smallest, for all four profile/benchmark pairs. One explanation is that with short-running applications this metric limits the number of methods that are candidates for optimization because fewer methods exceed the specified threshold. This implicitly limits the compilation overhead in short-running applications, but allows a larger set of methods to be optimized in longer-running applications.

Interestingly, the sample-based strategies are better than the time-based strategies (having a smaller average degradation from Best) for all but one case. This phenomenon could be partly due to inaccuracies in the time-based profile, however it can also be more fundamental in that method selection is inherently less error prone with a sample-based profile. Because the profile does not contain all executed methods, it automatically limits the selection to methods to those that are sampled at least once, essentially adding a secondary condition to the prediction strategy.[5]

In particular, "Samples in method" is the best overall strategy, performing very well for both the short- and longer-running benchmarks, and having its optimal values for both categories be very close (9 and 10). This is in contrast to all other short/longer strategy pairs, where the optimal values vary greatly. This result echoes the performance results obtained with an early version of the sampling-based Jalapeño adaptive system, which used an online version of "Samples in method" to guide optimization decisions.

[5] The sampled profile does not add a secondary condition to "Time/Samples in method" because it already limits the selection in exactly the same manner.

6 Related Work

Radhakrishnan et al. [28] used the Kaffe Virtual Machine to measure the performance improvement possible by interpreting, rather than compiling, cold methods for a subset of the SPECjvm98 benchmarks with input size 1. Although the two studies touch on some similar ideas, Radhakrishnan et al. focused on architectural issues for Java, while our study emphasized a comprehensive evaluation of selective optimization designed to aid the design and implementation of an adaptive system.

Kistler [23] presents a continuous program optimization architecture for Oberon that allows "up-to-the-minute" profile information to be used in program reoptimization. He concludes that continuous optimization can produce better code than offline compilation, even when past profiling information is used. Optimizations are 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.

The HotSpot JVM [22], the IBM Java Just-in-Time compiler (version 3.0) [29], the Intel JUDO system [14], and the Jalapeño JVM [5] are adaptive systems for Java. The first three systems initially interpret an application and later compile performance-critical code. The IBM JIT and Intel JUDO system uses method invocation counters augmented to accommodate loops in methods to trigger compilation. JUDO optionally uses a thread-based technique in which a separate thread periodically inspects the counters and performs compilation. Jalapeño uses a compile-only approach based on the same low-overhead sampling that is used in this paper. It employs a cost/benefit model to trigger recompilation. Details of the HotSpot compilation system are not provided.

Hölzle and Unger [20] describe the SELF-93 system, an adaptive optimization system for SELF. Method invocation counters with an exponential decay mechanism are used to determine candidates for optimization. Unlike the results of our study, they conclude that determining when to optimize a method is more important than determining what to optimize. We attribute this contrasting conclusion to the differing goals of the two systems. The goal of their system is to avoid long pauses in interactive applications, where as we focus on achieving optimal performance for short- and longer-running noninteractive applications.

Bala et al. [7] describe Dynamo, a transparent dynamic optimizer that performs optimizations on a native binary at runtime. 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 9%.

Burger and Dybvig [9, 10] 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 [19] 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.

Another area of research considers performing runtime optimizations that exploit invariant runtime values [6, 18, 26, 25, 15, 27]. 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 [17] and Self-91 [12] systems, as well many of today's JIT Java compilers [1, 24, 33].

There are also fully automated profiling systems that use transparent profiling to improve performance of future executions. Such systems include Digital FX!32 [21], *Morph* [34], and *DCPI* [4]. However, these systems do not perform compilation/optimization at runtime.

7 Conclusions

This paper describes an empirical study of selective optimization using the Jalapeño JVM. Two categories of benchmarks, short running (1.1–4.8 seconds) and longer running (16.6–70.6 seconds), are studied using SPECjvm98, pBOB, Volano, and the Jalapeño optimizing compiler as benchmarks. The main observations are

- The Best selective optimization strategy (counting compilation time) is within 67% of all-opt-exe for the short-running benchmarks and 14% for the longer-running benchmarks.
- Best substantially outperformed both none-opt and all-opt for the short-running benchmark suite, averaging 56% and 71% improvement, respectively.
- For the longer-running benchmarks, Best shows a much smaller improvement over all-opt, averaging 12%.
- The sweet spot is fairly small for short-running benchmarks, and large for longer-running benchmarks.
- Best from the incomplete, sample-based profile can match Best from the time-based profile independent of the application's running time and sweet spot size, even though the sample-based profile contains only a fraction of the executed methods.
- Predicting which methods to optimize is much harder for short-running applications than for longer-running applications. Predictions based on the sample-based profile are more effective than predictions based on the time-based profile. "Samples in method" is the most effective overall predictor, averaging 8.5% and 1.03% degradation from Best, respectively, for the short- and longer-running applications.

7.1 Impact on Adaptive Optimization

The goal of this work is to gain a better understanding of the issues surrounding selective optimization, and in doing so provide insight into the more general problem of adaptive optimization. Understanding the performance gains possible from selective optimization, as well as the classes of applications for which these gains can be expected, is useful for guiding efforts in designing an effective adaptive system.

As expected, selective optimization provides the largest potential benefit for the short-running benchmarks. The goal of an adaptive system will be to achieve as much of this win as possible.

The longer-running applications presented in this paper are approaching the point where all-opt is only slightly worse than Best, which, in turn, is only slightly worse than all-opt-exe. In addition, the large sweet spot of the longer-running benchmarks makes it easy to predict a set of methods to optimize that will result in reasonable performance. We feel this is encouraging for adaptive optimization because it suggests that with little effort an adaptive system can achieve close to the performance of all-opt-exe, and by concentrating its efforts on feedback-directed optimizations, such as inlining and specialization, can achieve performance beyond that of a static compiler.

The results of the sample-based profile are also encouraging. We now do not expect the accuracy of our coarse-grained sampler to be a problem for the adaptive system. The sample-based profile was more effective than the time-based profile when used to predict which methods to optimize, even for the short-running benchmarks, which run for as little as 1.5 seconds. This suggests that an adaptive system will be able to use an online, sample-based profile to make effective optimization decisions, and will be able to trust the profile after a short amount of time.

One concern is how an adaptive system should select methods for optimization. Even with the offline profile providing full knowledge of the future, the Best set of methods to optimize is difficult to predict for the short-running benchmarks. In an adaptive system, the problem becomes even more complex because less profiling information is available. Yet, unlike in this study, an adaptive optimization system is not required to make all of its optimization decisions at one time, and thus the predictors presented in Section 5 are not the only selection options available to an adaptive system. For example, an adaptive system could begin by optimizing only a small part of the application and continue until it is no longer improving performance. We see exploring such possibilities as an interesting area of future work.

Acknowledgments

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

References

1. Ali-Reza Adl-Tabatabai, Michał Cierniak, Cuei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 280–290, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), 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 Journal*, 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*, pages 314–324, 1999.
4. Jennifer M. Andersen, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? Technical Note 1997-016a, Digital Systems Research Center, www.research.digital.com/SRC, September 1997.
5. Matthew Arnold, David Grove, Michael Hind, Stephen Fink, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 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. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.
8. S. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1), 2000.
9. Robert G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme*. PhD thesis, Indiana University, 1997.
10. 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.
11. 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.
12. 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).

13. 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.
14. Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.
15. Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, January 1996.
16. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.
17. 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.
18. 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.
19. Gilbert J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs.* PhD thesis, Carnegie-Mellon University, 1974.
20. 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.
21. Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining emulation and binary transslation. *Digital Technical Journal*, 9(1):3–12, January 1997.
22. The Java Hotspot performance engine architecture. White paper available at http://java.sun.com/products/hotspot/whitepaper.html, April 1999.
23. Thomas P. Kistler. *Continuous Program Optimization.* PhD thesis, University of California, Irvine, 1999.
24. 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.
25. Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):1–5, September 1998. Article 23.
26. 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.
27. 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.
28. 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.
29. 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 Journal*, 39(1), 2000.
30. VolanoMark 2.1. `http://www.volano.com/benchmarks.html`.
31. John Whaley. Dynamic optimization through the use of automatic runtime specialization. M.eng., Massachussetts Institute of Technology, May 1999.
32. 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 Francisco, CA, June 1999.
33. 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.
34. 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)*, Operating Systems Review, 31(5), pages 15–26, October 5–8 1997.