Online Performance Auditing: Using Hot Optimizations Without Getting Burned

Jeremy Lau
University of California, San Diego
IBM T.J. Watson Research Center
jl@cs.ucsd.edu

Matthew Arnold
Michael Hind
IBM T.J. Watson Research Center
{marnold,hindm}@us.ibm.com

Brad Calder
University of California, San Diego
calder@cs.ucsd.edu

Abstract

As hardware complexity increases and virtualization is added at more layers of the execution stack, predicting the performance impact of optimizations becomes increasingly difficult. Production compilers and virtual machines invest substantial development effort in performance tuning to achieve good performance for a range of benchmarks. Although optimizations typically perform well on average, they often have unpredictable impact on running time, sometimes degrading performance significantly. Today's VMs perform sophisticated feedback-directed optimizations, but these techniques do not address performance degradations, and they actually make the situation worse by making the system more unpredictable.

This paper presents an online framework for evaluating the effectiveness of optimizations, enabling an online system to automatically identify and correct performance anomalies that occur at runtime. This work opens the door for a fundamental shift in the way optimizations are developed and tuned for online systems, and may allow the body of work in offline empirical optimization search to be applied automatically at runtime. We present our implementation and evaluation of this system in a product Java VM.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization, Run-time environments, Compilers, Code generation

General Terms Languages, Performance

Keywords Virtual machines, Feedback-directed optmizations, Java

1. Introduction

The goal of a compiler optimization is to improve program performance. Ideally, an optimization would improve performance for *all* programs, but some optimizations can also degrade performance for some programs. Thus, it is sometimes acceptable for an optimization to improve performance *on average* over a set of programs, even if a small performance degradation is seen for some of these programs. This often leaves aggressive optimizations, which can produce substantial performance gains and degradations, turned off by default in production compilers because it is difficult to know when to choose these optimizations, and the penalty for a wrong decision is high. Therefore, developing a compiler involves tuning a number of heuristics to find values that achieve good performance on average, without significant performance degradations.

Today's virtual machines (VMs) perform sophisticated online feedback-directed optimizations, where profile information is gathered during the execution of the program and immediately used during the same run to bias optimization decisions toward frequently executing sections of the program. For example, many VMs capture basic block counts during the initial executions of a method and later use this information to bias optimizations such as code layout, register allocation, inlining, method splitting, and alignment of branch targets [6]. Although these techniques are widely used in today's high-performance VMs [42, 50, 1, 33, 25], their speculative nature further increases the possibility that an optimization may degrade performance if the profile data was incorrect or if the program behavior changes during execution.

The empirical search community [9, 22, 55, 43, 31, 38, 52, 17, 37, 53] takes a different approach toward optimization. Rather than tuning the compiler to find the best "compromise setting" to be used for all programs, they acknowledge that it is unlikely any such setting exists. Instead, the performance of various optimization settings, such as loop unroll factor, are measured on a particular program, input, and environment, with the goal of finding the best optimization strategy for that program and environment. This approach has been very successful, especially for numerical applications. Architectures and environments vary greatly, especially for Java programs, and tuning a program's optimizations to its running environment is critical for high performance.

The majority of the empirical search has been performed *offline*. With the rich runtime environment provided by a VM, it becomes possible to perform fully automatic empirical search *online* as a program executes. Such a system could compile multiple versions of each method, compare their performance, and select the winner. Examples of such an online system are the Dynamic Feedback [21] and ADAPT [54] systems, and the work by Fursin et al. [24].

The most significant challenge to such an approach is that an online system does not have the ability to run the program (or method, etc.) multiple times with the exact same program state. Traditional empirical search, and optimization evaluation in general, is performed by holding as many variables constant as possible, including: 1) the program, 2) the program's inputs, and 3) the underlying environment (operating system, architecture, etc.). In an online system, the program state is continually changing; each invocation of a method may have different parameter values and different global program state. Without the ability to re-execute each optimized version of the method with the exact same parameters and program state, meaningful performance comparisons seem impossible. Previous online systems [21, 54, 24] do not provide a solution for general optimizations to address the issue of changing inputs or workloads

during the comparison. Because of this challenge, today's VMs make no attempt to apply any form of empirical search at runtime.

In this paper we present a technique that overcomes the challenge of evaluating the performance of multiple versions of an optimized method online in a production VM. The contributions of this work are

- a framework for a low-overhead completely automatic online system that evaluates the effectiveness of optimizations, dealing with changing program state;
- an offline feasibility study demonstrating the robustness of the statistical technique used in the framework;
- an online implementation of the framework in a production Java VM and an evaluation of its effectiveness in an online setting; and
- a sample client that demonstrates the potential performance improvements of using the framework.

Our techniques allow for online empirical optimization, greatly improving the ability of runtime systems to *increase performance* and *prevent degradations*.

The paper is organized as follows. Section 2 provides background and additional motivation for this work. Section 3 introduces our approach to online performance evaluation. After describing our methodology in Section 4, Section 5 presents an offline feasibility study that demonstrates the potential efficacy of this work. Section 6 describes our online implementation and evaluates it using a simple client. Section 7 discusses lessons learned from this work. Section 8 describes how this work compares to related work and Section 9 concludes this paper.

2. Background

This section presents further motivation and background for this work. Section 2.1 discusses two different approaches to performance tuning of optimizations: modeling and measuring. Section 2.2 provides empirical evidence to motivate why online performance evaluation is important.

2.1 Why Modeling Is Not Enough

In a traditional optimizing compiler, every transformation (or optimization) is fundamentally a performance prediction. For example, redundant load elimination makes the seemingly obvious prediction that removing load instructions will reduce execution time. Often, performance predictions are based on an abstract metric, with the assumption that there is a correlation between the metric and bottom-line performance. As another example, compilers often attempt to minimize the number of instructions executed along the critical path, relying on the assumption that fewer instructions results in faster execution.

Some of the compiler's performance predictions are made explicit through the use of a *performance model*. Examples of compiler optimization that have used explicit models are inlining, instruction selection, instruction scheduling, and register allocation. Inlining models the cost of an inlining decision (code growth, resulting in instruction cache pressure) weighed against its benefits (reduced call overhead and increased analysis scope); instruction selection models the cost of choosing various instructions; instruction scheduling models the instruction pipeline, the latency of instructions, and architectural hazards; and register allocation models the cost of spilling a register.

Implicit models are more prevalent than explicit models. The dozens of transformations and heuristics used throughout a compiler make performance predictions based on an implicit model. The compiler writer may not have explicitly formulated, or even fully understood the model, but it is still inherently hard-coded into the optimization. For example, many compilers contain an inlining heuristic that ensure methods larger than a certain size are never inlined; this simple rule can be considered an implicit performance model that predicts a lack of benefit (or potential performance decline) from inlining large methods.

Most optimizations that use profiling information (often referred to as feedback-directed optimization, or FDO) still employ a model, whether explicit or implicit. Although feedback-directed optimizations observe dynamic information, they generally do not measure program performance directly (wall clock time, etc.), but instead measure a dynamic metric, such as basic block frequencies or method invocation frequencies.[1] These dynamic metrics are then used as input to the optimization's model (usually an implicit model) to predict performance. For example, it is widely assumed that placing frequently-executed basic blocks close to each other will improve instruction cache performance, which will improve bottom-line performance. Optimizations are designed to assume that branch predictors will predict forward and backward branches in a certain way.

Although performance models are often effective, they do not always correlate with bottom-line performance. This results in performance degradations, which are common in a compiler's highest optimization levels. Section 2.2 provides empirical examples. Heuristics are often tuned extensively to avoid performance degradations, resulting in compromise settings that are not the best in any one configuration.

One way to address the shortcomings of performance models is to refine the models to more accurately predict performance, but we believe this is an impossible, never-ending task. *All* aspects of the execution stack must be correctly modeled, including the operating system, the hardware architecture and implementation, and even the compiler itself, as one optimization may interfere with another downstream optimization. With the current trend of adding more levels of complexity and virtualization to all levels of the execution stack, the problem will only become more difficult as true performance continues to diverge from the predictions of any model. Ten years ago it was not such a daunting task to model the performance impact of instruction scheduling; on today's hardware it is nearly impossible.

We advocate empirical search: to acknowledge that performance will be hard to model and predict, and to develop mechanisms that more directly detect performance mispredictions and react appropriately.

2.2 Examples to Motivate Searching the Optimization Space

This section provides evidence that demonstrates the difficulty of modeling the performance impact of optimizations. The first example looks at the effectiveness of optimization levels of a JIT compiler, and the second example looks at the effects of changing inlining heuristics.

Experimental Setup To motivate our work, we modified IBM's J9 JIT compiler to read the hardware cycle counter on entry and exit of selected program methods to measure the total number of cycles spent in a given method (further details on this methodology are described in Section 4). This infrastructure allows us to evaluate the impact of optimizations on individual methods in the application. To evaluate the quality of the code produced by the compiler, these experiments measure *steady-state* performance, which is a configuration of the benchmark that excludes the early execution of a program when all (dynamic) compilation would occur.

For the remainder of the paper, we focus on what we will refer to as the *hot* methods of the benchmarks (further details about the benchmark suite and our selection of hot methods are presented later in Section 4). In our 20 benchmarks, there are 101 hot methods.

[1] Notable exceptions include the work of Adl-Tabatabai et al. [2], where information from hardware performance monitors is used to trigger dynamic optimizations.

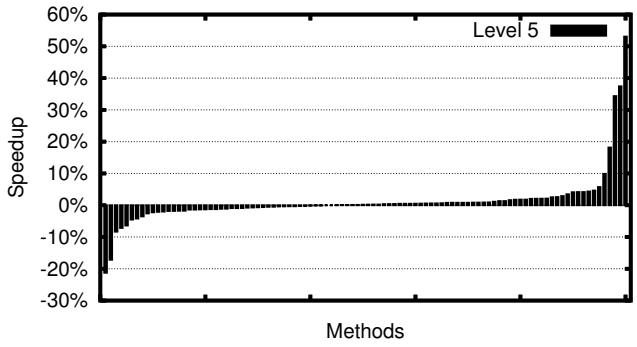

Figure 1. Per-method performance impact of moving from optimization level 4 to level 5.

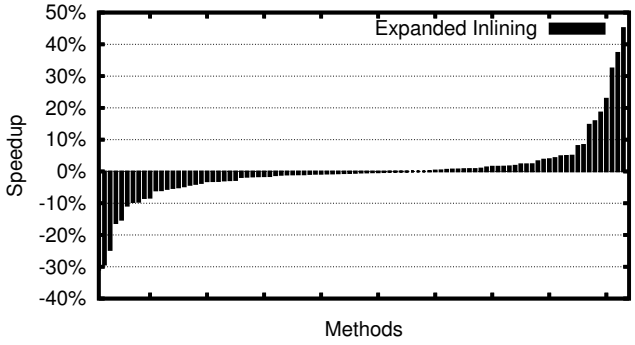

Figure 2. Per-method performance impact of expanded inlining heuristic.

Optimization Levels Most compilers group optimizations into *levels*, where higher levels provide better performance at the cost of higher compilation time. In a JIT setting, a particular level is chosen whenever a method is compiled.[2] The J9 JIT has five such levels, which we will refer to as O1–O5.

We used our timing infrastructure to compare the performance of the two highest levels: O4 and O5. On average, across the entire benchmark suite, level O5 improves performance by 1.5% over level O4 with a maximum improvement of 12.5%, and does not significantly degrade performance for any benchmark. However, when measuring performance of O4 versus O5 for individual methods, the results vary substantially.

Figure 1 compares the performance of O4 and O5 for each of the 101 hot methods. Each bar represents a method, and the y-axis reports the percentage speedup of O5 over O4 (higher means that O5 is faster). As is typical in compiler optimizations, the highest level (O5) offers substantial speedups for a small number of methods (more than 10% speedups for the 5 rightmost methods), with modest effects on most of the methods.

However, level O5 actually *degrades* performances relative to O4 for about a third of the methods, and the leftmost method is degraded by 21%. This result is not a byproduct of a poor performing JIT compiler; J9 performs competitively with industry leading JVMs and substantial time and effort has gone into tuning its optimization levels to maximize performance. The results are an inevitable byproduct of tuning compiler heuristics for the *average* case, and evaluating performance at the program level. We expect that this result can be reproduced with any optimizing compiler.

Method Inlining Our second study looks at a specific optimization, method inlining. Conceptually, method inlining has a fairly clear cost-benefit model. The potential benefit of inlining is the performance gained by removing call overhead, and optimizing the callee in the context of the caller. The potential cost of inlining is a decrease in instruction cache locality, and increasing the pressure on downstream optimizations such as register allocation. These performance effects are nearly impossible to predict accurately, so the inlining component in optimizing compilers typically contains dozens of heuristics and ad-hoc tuning knobs. These heuristics are often tuned on large benchmarks suites to find values that work well, on average for these benchmarks.

To demonstrate the difficult task of predicting the performance of method inlining, we expanded one of the heuristics substantially, and evaluated the impact. For this experiment we quadrupled the threshold that limits the size of a callee that can be considered for inlining; thus the new heuristics allow inlining to be performed for substantially larger callees. Figure 2 shows the performance of the new threshold relative to the original inlining thresholds, for the hot methods in our benchmarks. As in Figure 1, each bar represents a method, and the y-axis reports the percentage speedup with expanded inlining thresholds. As in Figure 1, the impact of expanded inlining varies greatly among methods. On the right side of Figure 2, we see 7 methods that improve by over 10%, 4 of which are over 20%. On the left side of the figure, there are 5 methods that degrade by over 10%, with 2 larger than 20%. Reducing the inlining thresholds produced similar, inconsistent performance across the set of hot methods.

These large variations are a compiler writer's nightmare, because the higher threshold offers great promise, but also runs a significant risk of performance degradation. However, the variations illustrate the potential for a technique that can automatically find the tuning values that give performance improvements, while avoiding those values that degrade performance.

3. The Performance Auditor

This section introduces the *Performance Auditor*, a framework that enables online performance evaluation. The framework tracks the bottom-line performance of multiple implementations of a region of code to determine which one is fastest. The analysis is performed online: as the application executes, performance evaluations are done and the results are immediately available. No offline training runs or hardware support are required.

As further described in Section 6, this work uses method boundaries to delimit code regions (as is common in virtual machines) and focuses on comparing only two versions of each method at a time; however, the ideas presented are not limited to these design choices. We refer to the process of identifying the faster method variant as a performance *bakeoff*. The component that requests and uses the bakeoff results is called the *client*.

A client of our framework is responsible for 1) providing the alternative optimized implementations of the code region to be compared and 2) deciding what action to take with the outcome of the bakeoff. One example client of this framework is to evaluate aggressive optimizations that significantly improve performance for some code regions, but degrade performance for others. By using our framework, one can identify the performance degradations and revert to the original implementation. Another example client could empirically search the tuning space for important optimization parameters, such as the inlining thresholds described in Section 2.2, and choose the parameter that gives the best performance.

[2] Because of Java's dynamic nature, traditional static interprocedural analysis is not performed because the complete call graph for the program's execution is not known until the program terminates [27].

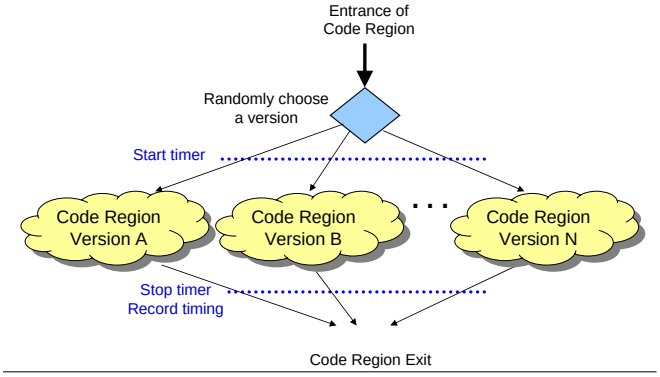

Figure 3. Performance Auditor Overview

Our solution has two components: 1) a technique to collect execution times of an implementation of a code region, and 2) a statistical mechanism to determine which set of timings is fastest.

Figure 3 presents a high-level view of our approach. The client provides N optimized versions of the code region to evaluate. The key idea is to randomly select one of these implementations whenever the code region is entered, and record the amount of time spent. As the program executes, timings for each implementation are collected for later analysis.

The second component of our framework analyzes the collected timings to determine which is fastest. The biggest challenge is that the executions most likely occur with different program state (parameters and/or memory values). This implies that the different implementations of this code region could have taken different paths and operated on different data during their executions. For example, a particular execution of version A of the code region may run for less time than version B of the code region, not because A has a better implementation, but because an argument to the method had a value that resulted in A doing less work. For example, if the code region is a sorting algorithm, version A may have been invoked with a list of 10 elements, while version B was invoked with a list of size 1,000.

We overcome this challenge by leveraging the law of large numbers. Specifically, as the framework accumulates more timing samples for versions A and B, variations in timings will be evenly distributed to the two versions if the versions are selected randomly to avoid any correlation with the program's natural behavior. Given the timing data, the framework performs a statistical analysis and concludes which version is faster, with some degree of confidence.

Our system decides which version (A or B) is faster by averaging the timing samples for each version. By comparing the averages, the system can determine which version is faster, but the comparison is meaningful only if there are enough timing samples. To check if there are enough samples, we define a confidence metric that calculates the probability that the actual performance difference between versions A and B is consistent with the observed performance difference in the timing samples from versions A and B.

To calculate our confidence metric, the mean and variance is computed for each set of timings. Then the absolute difference between the means $|mean(A) - mean(B)|$, and the variance of the difference between the means $\frac{variance(A)}{sizeof(A)} + \frac{variance(B)}{sizeof(B)}$ is computed. This computed mean and variance define a normal probability distribution, and our confidence value is computed by integrating this distribution from 0 to ∞. We use the absolute difference between the means, which is always positive, so this confidence value is the probability that the difference between the means for the entire dataset is consistent with the difference between the means for our samples.

The number of samples needed before a confident conclusion can be made is directly correlated with the variance in the amount of work performed by the code region. If the code region performs a constant amount of work on each invocation, a small number of samples will be needed to conclude that the performance difference is statistically significant. As the variance of the code region's running time increases, so does the number of samples needed before a confident conclusion can be made.

For this approach to succeed, we must ensure that our timings do not correlate with program behavior, because correlations can bias the timings for one version of the code region. If there are no correlations, we should be able to identify the performance differences among the versions after enough samples are collected.

Sections 5 and 6 explore this concept in more detail. Section 5 investigates the feasibility of our hypothesis through an offline study, and Section 6 describes the online system.

Program	Methods Executed	Bytecodes Exe (KB)	Hot Methods
antlr [51]	1702	228	2
bloat [51]	1063	105	7
compress [48]	770	66	2
daikon [18]	2108	171	6
db [48]	782	67	3
hsqldb [51]	1416	147	8
ipsixql [15]	828	61	6
jack [48]	746	56	3
javac [48]	1467	133	3
jbb2000 [47]	1197	115	8
jess [48]	1140	86	4
jython [51]	1777	186	15
mpegaudio [48]	866	78	4
mtrt [48]	853	76	4
phase [40]	450	31	2
pmd [51]	2030	128	4
ps [51]	946	75	3
soot [46]	2061	235	4
xalan [51]	2108	171	5
xerces [56]	521	36	8

Table 1. Benchmark suite

4. Methodology

The experiments in this paper were performed using a development version of IBM's J9 VM and its high-performance optimizing JIT compiler [25]. The VM was run on a Pentium 4 3.0 GHz machine with 2 processors and 1 GB RAM running Linux. We use a suite of 20 benchmarks composed of the complete SPECjvm98 benchmark suite [48], the SPECjbb2000 benchmark [47], and several other Java applications, including seven of the DaCapo benchmark suite [51][3]. Table 1 reports the number of methods executed,[4] as well as the total size (in KB) of all bytecodes executed for each benchmark. These numbers report dynamic metrics, i.e., they are based on what is executed, not what could be executed.

The last column gives the number of "hot" methods for each benchmark. We define the hot methods as methods that consume enough cycles to be selected by J9 for aggressive feedback-directed optimizations. For these methods, J9 first performs an additional compilation to instrument the method for profiling, then optimizes the method at one of the two highest optimization levels (O4 or O5).

To collect timing samples for a compilation of a method, we use the Pentium 4's Read Time-Stamp Counter (RDTSC) instruction. The RDTSC instruction reads the processor's 64-bit cycle counter

[3] The development version of the VM that we used did not run the remaining three benchmarks from the DaCapo suite.

[4] Number of methods executed includes all Java methods (including library methods) executed by the JVM while loading and executing the application.

and stores its value in a register. Method entry is instrumented to read the cycle counter and store the cycle count in the method's stack frame. Method exit is instrumented to read the cycle counter again, subtract the current cycle count from the cycle count on the stack, and store the difference in a circular buffer.

This methodology measures the total time that the method was active on the stack to ensure fair comparisons in the presence of changing inlining decisions. Measuring only time spent in the instrumented method (excluding callees) would produce incorrect results in the presence of method inlining.

5. Offline Convergence Study

This section presents an offline feasibility study that demonstrates the potential efficacy of this work. Section 5.1 describes how timing samples are collected and used to evaluate our approach. Section 5.2 considers using a fixed number of samples, and shows that thousands of samples are needed to accurately detect speedups during a bakeoff. Section 5.3 then shows that our confidence analysis can be used to correctly detect a performance difference between two different compilations of the same method.

5.1 Experimental Setup

Our goal is to determine how many samples are needed to accurately detect a performance difference between two compilations of a method. To quantify this, we time each invocation of each hot method. We use these timings to represent two hypothetical implementations of the hot method. For a hot method, half of the samples are randomly assigned to set A, and the other half to set B, and each sample in set B has an artificial speedup of $X\%$ applied to each of its samples. A and B then represent two potential implementations of that method: A is the baseline, and B is an optimized version that runs exactly $X\%$ faster. We can then determine how many samples are needed to recognize that B is faster than A.

For this analysis, we gather method timing samples for the hot methods in each benchmark. For these experiments we stop gathering samples for a method if the method has run for 2 minutes of CPU time. After collecting a set of method invocation timings, we sort the timings and eliminate the highest 10%. This filters out noise introduced by garbage collection or context switches. Next, we randomly divide our pool of timing data into two disjoint sets (A and B), each containing half of the timings collected, and we artificially decrease each cycle count in set B by $X\%$.

5.2 Fixed Number of Samples

Prior online systems that performed a bakeoff between different compiler optimizations took either one [21, 54] or two [24] timing samples for each optimized version of the code. These systems determined which version was faster based on that sample. In our more general setting, we found that thousands of samples are needed to accurately determine which version is better with our workloads.

To show this, we experiment with a simple approach that takes a fixed number of samples and decides which is faster. We use the sets A and B as described above, and we randomly select a fixed-size subset A' of A, and a equally sized random subset B' of B. We then compute the means of the times in subsets A' and B', and if the mean for B' is less than the mean for A' we report a correct prediction. If not, we report a misprediction. To reduce noise due to our use of randomness, we repeat each experiment 100 times with different random seeds and average the results.

Figure 4 shows the number of mispredictions when 10, 100, 1000, and 10000 samples are used for A' and B'. The x-axis shows the speedup introduced in set B, and the y-axis shows the percentage of experiments in which we incorrectly predict that A is faster than B. For this graph, we use method invocation timings, as described in the prior section.

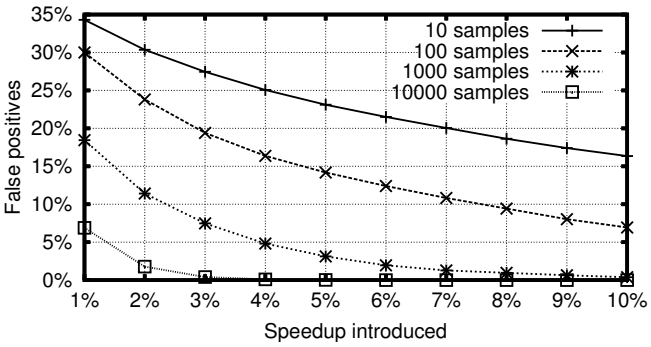

Figure 4. Misprediction rate for a simple sampling approach in which a fixed number of method invocation (entry+exit) timings are collected, for various amounts of introduced speedup.

Figure 4 shows that a large number of samples must be collected to detect small speedups, and fewer samples are needed to detect larger speedups. For example, to reliably detect a 2% speedup, we need at least 10,000 samples, but 1,000 samples are adequate to detect a 10% speedup. The results show that we can detect both large and small speedups if we always collect 10,000 samples, but we need to only collect that many samples to detect small speedups — 1,000 samples can detect a 10% speedup in an order of magnitude less time, with only 0.4% additional mispredictions. This motivates a statistical approach that can efficiently detect both large and small speedups by collecting only as many samples as needed, which we examine next.

5.3 Statistical Approach

We now evaluate our confidence-based technique described in Section 3 to predict if B is faster than A. We start by establishing sets A and B, as described above.

To determine the number of samples necessary to confidently predict a performance difference between A and B, we first set a confidence threshold $Z\%$ (80% and 99.99% in this study). We run our experiment by adding samples to A' from A, and to B' from B until we are $Z\%$ confident in our performance prediction, based on the formulas in Section 3.

We start with 100 samples each in A' and B', and perform the confidence evaluation. If the confidence is not above our threshold, we add 100 more samples to both A' and B', and repeat. We continue increasing the size of the subsets until the confidence is above our threshold. When we perform these experiments, we may run out of samples before we reach our confidence threshold. When this happens, we report that the method did not converge, and we cannot make a performance prediction. We repeat each convergence experiment 100 times with different random seeds and average the results.

When we are confident that we can predict a performance difference, we report the time to converge, which is the sum of the cycle counts in subsets A' and B'. For these confident predictions, if we (incorrectly) predict that A is faster than B, we report a misprediction. If we run out of samples before we reach our confidence threshold, we report that the experiment did not converge.

Figure 5 shows the percentage of experiments that converged when an artificial speedup is introduced in B. The y-axis shows the percentage of hot methods for which we generated predictions for the two confidence levels examined. The `entry+exit` results use the per method timing, and the `yieldpoints` results will be described shortly. These results show that it is more difficult to detect smaller speedups in 2 minutes of CPU time for our hot methods, if high confidence is desired.

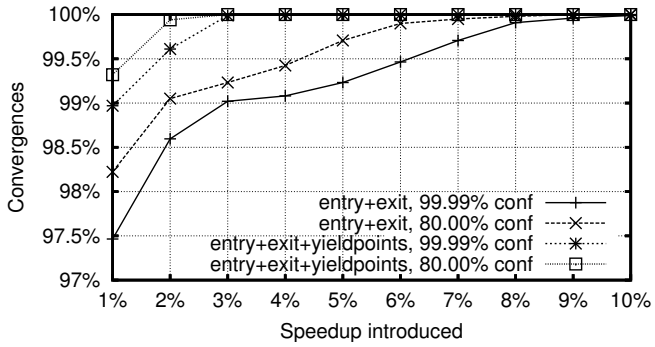

Figure 5. Convergence rate for our statistical approach. This plot shows the percentage of hot methods for which we generated confident predictions, given our sampling limit of 2 minutes of CPU time for each method.

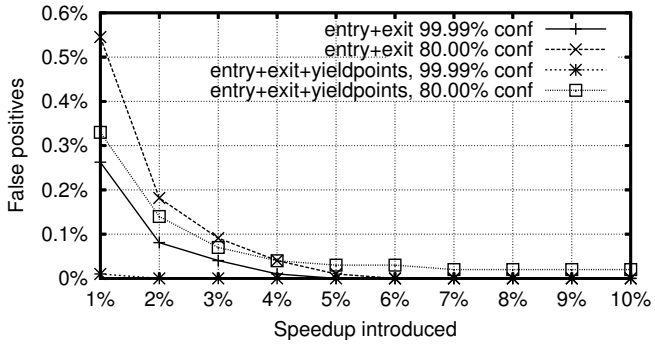

Figure 6. Incorrect predictions. This plot shows the percentage of hot methods in which we (incorrectly) predict that A is faster than B. Results are averaged over converged methods.

Figure 6 shows the percentage of experiments in which we incorrectly predicted that A was faster than B. The results in this figure and figure 5 show the tradeoff between the number of accurate predictions and the number of confident predictions. As expected, incorrect predictions occur more frequently when speedups are small.

Figure 7 shows the number of cycles needed to detect a performance difference. The y-axis shows the total amount of time needed to make a confident performance prediction. For experiments that did not converge, we use the total time spent before giving up. In other words, this graph includes experiments that did not converge, and for those experiments, it shows a lower bound on the convergence time. The results show that it takes significantly longer to detect small speedups than large speedups. The results also show that by adjusting our confidence threshold, convergence time can be further reduced by sacrificing accuracy.

5.3.1 Using Yield Points for Timing

We also consider collecting timing samples at every loop branch. Collecting one timing sample from each method invocation works well for methods that are invoked frequently, but infrequently invoked methods may pose a problem because they generate timing samples more slowly, which means it will take longer to detect a performance difference. To address infrequently invoked methods, we consider a scheme in which we collect many timings from a single method invocation by collecting timing samples between temporally adjacent pairs of yield points. A yield point is a compiler-inserted statement that checks if the application needs to be temporarily suspended to perform a virtual machine service, such as garbage collection. The compiler inserts yield points on loop back-edges, so this approach approximates collecting timing samples for each loop iteration. If a hot method is infrequently invoked, that method is most likely spending a lot of time in loops, which we can measure by timing between yield points.

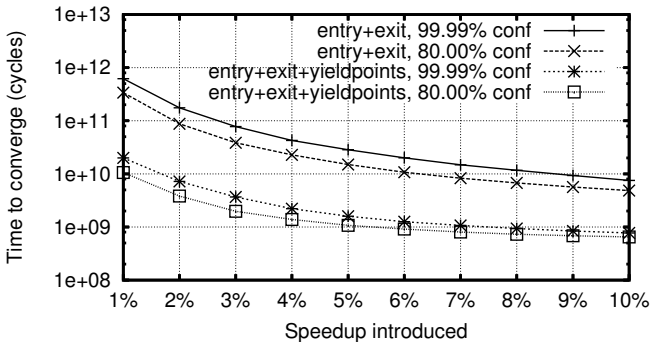

Figure 7. Time to converge. This plot shows the number of cycles needed to make a prediction. This graph includes experiments that did not converge, and for those experiments, it shows the amount of time spent in the experiment before giving up, which is a lower bound on the convergence time.

Figures 5, 6, and 7 show results with yield-point timings. These results suggest that collecting many timings per method invocation by instrumenting yield points reduces false positives as well as the time to converge, making it an attractive option. However, this offline study was conducted with simulated speedups applied to method timings. Collecting yieldpoint timings when comparing differently optimized versions of code is more challenging, as it requires keeping the yieldpoint placement consistent in both optimized versions. For example, if inlining occurs in the method being timed, it may inline additional yieldpoints into the method, resulting in more frequent samples. If any instructions can be identified uniformly in both optimized version of the code, they would be candidates for timing instrumentation. A second problem is that instrumenting frequently executed instructions, such as loop branches, can introduce more overhead in an online system. Careful engineering can be used to reduce this effect, such as timing groups of N yieldpoints, instead of timing each temporally adjacent pair of yieldpoints.

For the remainder of the paper we use method invocation timings with method entry and exit instrumentation, even though more time is needed to make a prediction. Collecting timing samples between yield points is a promising approach with challenges that we leave to future work.

6. Online Performance Auditor

This section describes the implementation of the online system. Before describing the system, we summarize some relevant background on adaptive optimization in virtual machines. We then present our implementation, which is composed of two parts: 1) a dispatch mechanism to select which version of code to run, and 2) a background thread to process and analyze the timing data collected. This is followed by an empirical evaluation of our online technique with a sample client.

6.1 Adaptive Optimization in Virtual Machines

All high-performance VMs use a selective optimization strategy, where methods are initially executed using an interpreter or a non-optimizing compiler. A coarse-grained profiling mechanism, such as method counters or timer-based call-stack sampling, is used to find

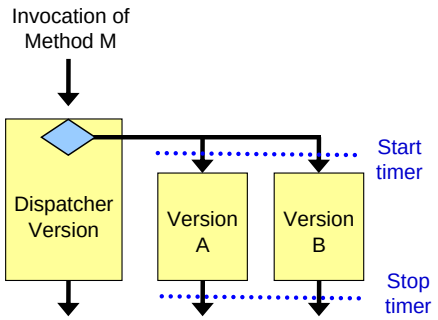

Figure 8. Architecture for the *dispatcher*, to select method invocations for timing.

```
METHOD ENTRY:
  sampleCountdown --;
  if (sampleCountdown < 0) goto BOTTOM;

  ... body of method M ...

BOTTOM:
  if (sampleCountdown < (BURST_LENGTH * -1)) {
    // Burst completed. Toggle and reset count
    toggle = !toggle;
    sampleCountdown = SAMPLE_COUNT_RESET;
  }
  if (toggle)
     return invokeVersionA();
  else
     return invokeVersionB();
```

Figure 9. Dispatch logic

hot methods. These methods are then dynamically compiled using the JIT compiler.

VMs use the JIT compiler's multiple optimization levels to trade-off the cost of high-level optimizations with their benefit; when a method continues to consume cycles, higher levels of optimization are employed. Some VMs, like J9 [39], will compile the hottest methods twice: once to insert instrumentation [7] to gather detailed profile information about the method, and then a second time to take advantage of the profile information after the method has run for some duration. Thus, in modern VMs a particular method may be compiled many times (with different optimization strategies) during a program's execution. Overhead is kept low by performing such compilations on only a small subset of the executing methods, and often by compiling concurrently using a background compilation thread.

6.2 Method Dispatcher

As mentioned in Section 3 we use methods as code regions for each performance bakeoff. Thus, we must intercept the method dispatch for the method being timed (M) so that execution can jump to an implementation at random. There are a number of ways this could be implemented in a VM. We chose to implement the dispatch as shown in Figure 8, where a full method body is compiled to act as the dispatcher. The dispatcher method is not timed, but contains a conditional test to decide whether a timed method should be executed, and if so, to invoke the correct one.

Compiling a full method body for the dispatcher is not necessary, but doing so makes it easy to reduce the overhead of the timing instrumentation, because only a subset of the calls to M jump to one of the instrumented versions. In situations where compiling a third version of the method is not feasible, alternative implementations could exclude the method body from the dispatcher so it always selects one of the timed methods. The dispatch logic could also be placed at the beginning of one of the timed methods, or inlined at the method's call sites.

Figure 9 shows the dispatch method logic. The prologue checks a sampling condition to determine if any timed version should be executed. We use a count-down sampling mechanism, similar to Arnold and Ryder [7]. This makes the fast-path of the dispatcher method reasonably efficient, containing only a decrement and check. Once the sample counter reaches zero, it jumps to the bottom of the code to determine which timed version should be executed. This code is on the slow path and can afford to execute slightly more complicated logic.

To minimize timing errors caused by caching effects from jumping to a cold method, the dispatcher invokes the timed methods in consecutive bursts. Once the burst threshold ($BURST\_LENGTH$) number of invocations of the timed method has occurred (tracked by letting the sample counter go negative) the sample counter is reset. A toggle flag is maintained to toggle between optimization versions A and B. If more than two versions are being compared, this toggle can easily be replaced by a switch statement. All of the counters are thread-specific to avoid race conditions and ensure scalable access for multi-threaded applications.

If a pure counter-based sampling mechanism is used, it could correlate with program behavior, and even a slight correlation could skew the timing results. One solution would be to randomly set the toggle flag on each invocation. Instead we vary the sampleCounter and burst length by periodically adding a randomly selected epsilon. In our system we use the VM sampling thread to update these quantities every 10ms, the fastest interval available in a default Linux kernel, and this is sufficient to avoid deterministic correlations.

6.3 Data processing

As the method timings are collected, they are written into a circular buffer (one per thread). This data is processed by a background thread that wakes up periodically and scans through each thread's buffer every 10ms. The processing thread starts from the position in the buffer where it left off last time, and scans the buffer until it catches up with the producer. No synchronization is used, so race conditions are possible, but dropping a sample occasionally is not a concern. The system is designed so that races cannot corrupt the data structures; slots in the buffer are zeroed after being read to avoid reading a buffer twice if the consumer skips past the producer.

While processing the buffer, there are two primary tasks: 1) discard outliers, and 2) maintain a running average and standard deviation. Outliers cannot be identified by collecting and sorting the entire dataset because this would require too much space and time for an online system. Instead, the system processes the data in smaller chunks of N samples, discarding outliers from this smaller set of times. The outliers are identified as the slowest M times in the chunk. Maintaining the running average and standard deviation is trivial by keeping a running total of the times, as well as the sum of the squares.

Our system also requires that a minimum number of samples are collected (10,000) before confidence analysis is performed, to ensure that a small initial bias in the program sampling does not lead to incorrect conclusions. Once the sufficient number of data points are collected, the confidence function described in Section 3 is invoked periodically to determine if the difference between the means is statistically meaningful. If a confident conclusion is reached, the bakeoff ends and the winner is declared; otherwise, the bakeoff continues. To ensure that a bakeoff does not run indefinitely, the system can also end the bakeoff after a fixed amount of wall clock time, or execution time, has expired.

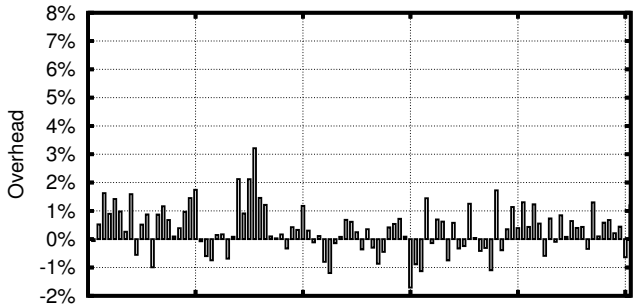

Figure 10. Per-method overhead of the dispatcher fast-path. No timing samples are being taken.

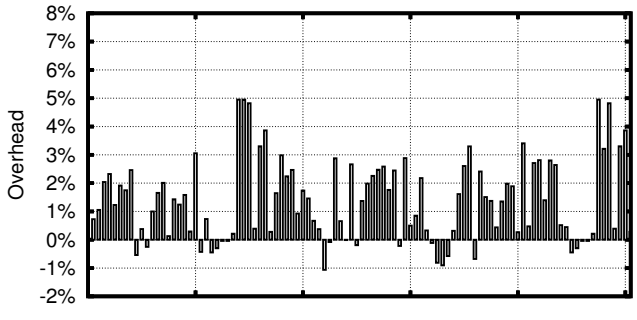

Figure 11. Per-method overhead of the Performance Auditor when sampling 1 of 20 executions. Overhead includes recording and processing the timing samples.

6.4 Overhead

There are three primary sources of overhead for our technique: 1) executing the sampling condition on the fast path of the dispatcher, 2) reading and storing the processor's cycle counter at method entry and exit, and 3) processing the timing samples with the background thread. This overhead is for the infrastructure itself, and is independent of the optimizations being compared.

To quantify the overhead, the online system was configured so that it constantly performs a bakeoff. Our online system will perform one bakeoff at a time, so we evaluate the overhead for each of the hot methods in our data set (as described in Section 4) independently.

Figure 10 presents the overhead incurred when a large sample interval is used so that effectively no samples are taken. This data primarily represents the overhead of the fast-path sampling check in the dispatcher. This overhead is quite low, averaging 0.4%, with all but three methods less than 2%. This result is important because it shows that the overall overhead of the auditor can be reduced to this value by lowering the sample rate. Negative overhead is most likely noise, which is expected when measuring overhead in the range of 1%.

Figure 11 presents the overhead when the dispatcher samples 1 out of 20 invocations of the instrumented method. This sample rate is fairly aggressive to allow quick convergence. It is the sample rate used in our full online system. The overhead increases when samples are taken, with an average of 1.5%, with some methods up to 5%. This amount of overhead is likely to be acceptable as it is incurred only during the bakeoff period. If lower overhead is desired, the sample rate can be reduced and the bakeoff can be performed over a longer period of time.

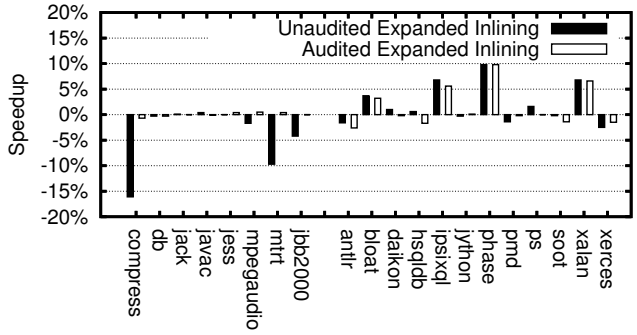

Figure 12. Performance of the online system using our technique to guide inlining heuristic selection.

6.5 Inlining Client: Full Online Optimization

This section describes our instantiation of a fully automatic online system that uses our framework to improve performance. As described in Section 6.1, the J9 VM already performs an additional compilation of the hottest methods to insert profiling instrumentation. We modified the recompilation logic to perform a bakeoff for these hot methods after they have been instrumented. When the bakeoff has completed, if a winner is confidently identified, the winner's optimization parameters are used for the final compilation of the method; otherwise, the VM's default behavior is used. We used the inlining example presented in Section 2.2 as an optimization client, comparing 1) the original inlining heuristics, and 2) the inliner with the size thresholds quadrupled.

The current system performs only one bakeoff at a time, in the order that methods are selected for instrumentation by J9. If a method is selected for a bakeoff while another bakeoff is already in process, it is added to a queue of pending bakeoffs. Multiple bakeoffs could potentially be performed simultaneously, but we did not experiment with this strategy.

Performance Figure 12 presents the steady-state performance achieved by our system. The x-axis presents the benchmarks, and the y-axis represents performance improvement relative to the default J9 VM. The black bar shows the performance when using the expanded inlining heuristic for *all* hot methods; the white bar shows the performance of the performance auditing system that runs bakeoffs to choose the default or expanded inlining heuristics.

Using the expanded heuristics without auditing resulted in performance improvements between 5–10% for three of the benchmarks in our suite (ipsixql, phase, and xalan), but large degradations for two benchmarks (compress and mtrt). When using auditing, our system was able to achieve most of the performance wins of the expanded heuristics, while avoiding the significant degradations.

The primary contribution of our work is not the speedup produced by this particular optimization client, but the performance auditor's success in identifying the better-performing version. The most important aspect of this performance result is that the system did *not* degrade performance measurably for any of the benchmarks in our suite, which demonstrates the viability of this technique to exploit high-risk/high-reward optimizations.

The left group of 8 benchmarks in Figure 12 are from the Standard Performance Evaluation Corporation (SPEC); these benchmarks are used widely within the industry for performance benchmarking, and most commercial JVMs have been tuned heavily for these benchmarks. It is therefore not surprising that our simple inlining heuristic adjustment did not improve their performance. However, when executing new benchmarks, such as the 12 benchmarks on the right, some substantial performance opportunities were discovered. We do

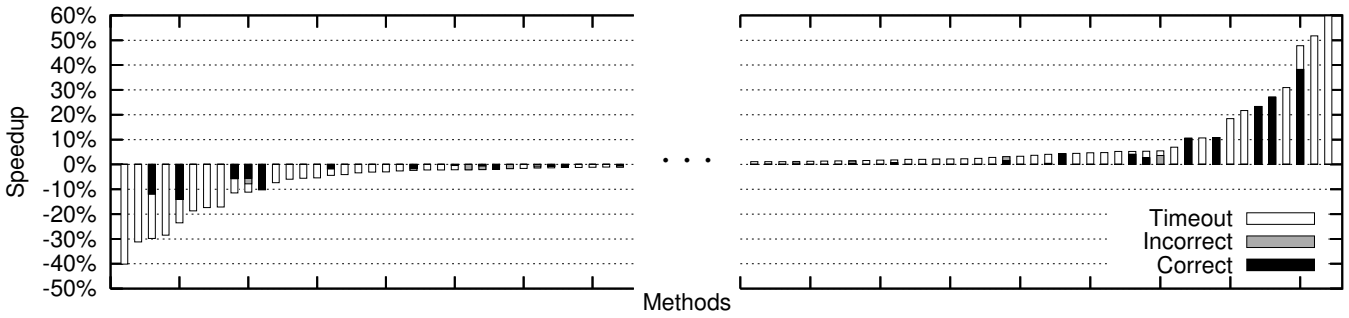

Figure 13. Accuracy of the online system using our technique to guide inlining heuristic selection.

not believe this to be an anomaly, but another example of the fundamental issue addressed by this work: predicting performance for unseen programs is difficult, and there is tremendous opportunity available if systems can automatically identify and correct performance anomalies.

Our auditing system has little impact on initial program startup behavior, because the performance auditing system only operates on methods that reach the highest levels of optimization. When a bakeoff is performed, the method is compiled 3 times, then once again after the bakeoff completes. Our current prototype makes no effort to distribute these compilations over time to avoid overhead, and thus, may introduce overhead relative to the original system before reaching steady state. This source of overhead can be avoided fairly easily by using known techniques, such as distributing the compilations over time using a low priority background compilation thread, or moving compilation to an additional processor if available. However, the primary goal of this work is to evaluate the potential of performance auditing by evaluating the accuracy of the bakeoff mechanism, and its potential impact on steady state performance.

Accuracy To evaluate the accuracy of our technique, the bakeoff decisions from the online auditing system were compared to the performance results from offline measurements. When an online bakeoff is performed, there are three possible outcomes:

1. **Timeout:** The bakeoff was terminated because it executed for too long without reaching a conclusion.
2. **Incorrect:** The system made a conclusion that was inconsistent with the offline measurements.
3. **Correct:** The system made a conclusion that was consistent with the offline measurements.

There are many sources of potential nondeterminism in our auditing mechanism, as well as in the underlying VM itself, so we ran the online auditing system 10 times for each benchmark, and the results for all bakeoffs were recorded.

Figure 13 presents the accuracy of the decisions made by the online system. The graph is the same format as Figure 2; each bar represents a hot method from our benchmark suite, and the total height of the bar represents the performance improvement (or degradation) caused by the expanded inlining heuristics for that method *reported by our offline measurements*.

Each bar is broken down by shades of gray to show the results of the online bakeoffs for that method. A solid black bar means that the correct conclusion was reached every time a bakeoff was performed. A bar that is 50% white means that 50% of the time the bakeoff ran too long and timed out. Methods where expanded inlining impacted performance by less than 1% were excluded from the graph to focus on the remaining methods. Bakeoffs were conducted for 152 methods; 73 methods impacted performance by less than 1%, and the remaining 79 are presented in Figure 13.

The performance of the excluded methods is small enough that a) the accuracy of the decision is irrelevant to overall performance, and b) the accuracy breakdown was not visibly discernible from the figure.

The most important aspect of our system is how accurate it is on the methods for which the expanded heuristics have a large impact, either positive or negative. Of the methods that show greater than 10% speedup from the expanded heuristics (rightmost bars), our online system makes the correct decision for around half of these methods (5/11) almost every time; the remaining 6 methods result in a timeout on every bakeoff. The system makes no wrong decisions for these methods.

Similarly, for the methods that expanded heuristics result in degradation of 10% or more (leftmost bars), our system often makes correct decisions for 5 of the 11 methods, and consistently times out on the remaining methods. For one method, the wrong decision was made in 2 of the 10 bakeoffs.

The most important point from this data is that the incorrect decision was avoided in almost all cases for the methods with the biggest potential gains or loses from the optimization. As shown by the performance results earlier in this section, making correct conclusions on a subset of the hot methods can result in substantial performance improvements at the program level.

The large number of timeouts is because of the time-to-convergence problems described in more detail in Section 7. Recall that the system collects only one timing sample each time the profiled method is invoked, so many methods do not generate timing samples fast enough to make a confident performance prediction before the bakeoff times out. These methods (approximately 50% of the bakeoffs) are a lost optimization opportunity. However, because our system uses the default optimization strategy in these cases, the resulting steady state performance will be the same as if no bakeoff was performed.

7. Discussion

The approach taken by this paper — comparing the performance of multiple versions of a region of code *without* attempting to hold inputs and program state constant — is often initially dismissed as infeasible. Our experiences with this project lead us to believe that the technique is in fact feasible, but it has different challenges than originally thought.

The biggest challenge is time to convergence, i.e., the number of data points that must be collected before making an accurate conclusion. Regions of code with high timing variance will eventually converge if the variance is finite, but the number of samples required may be impractical. Terminating the bakeoff early due to a timeout is not a serious problem in our system because the default optimization can be used; however, this means the resources used to perform the bakeoff were wasted, and an optimization opportunity may have been lost. Reducing convergence time is an important area of future

work, and we believe the approach of timing at a finer granularity (such as timing paths instead of method invocations) is a promising direction.

One technical challenge was engineering the system to ensure that the collection and processing of data did not skew the results. Whenever possible, our system randomizes the order in which data is collected and processed (e.g., which version of the code is timed first, which buffer is processed first, etc.). Adopting this approach whenever possible had an impact on the accuracy of the decisions made by the online system.

Removing outliers from the timing sets was also a key to timely convergence. A VM environment has many sources of timing noise, and removing outliers from the timings was an effective solution. Removing outliers is not strictly necessary, because the noise would be evenly distributed across all optimized versions; however, removing the outliers improves convergence time substantially. Unfortunately, removing outliers also has a downside; some data points labeled as "outliers" could have been a legitimate effect of the optimization being evaluated. For example, an optimization that increases the cost of a rare path might not be detected if too many outliers are discarded.

A key to reducing the number of discarded outliers is to reduce the noise in the timing mechanism itself. One source of noise is when timings are polluted by VM activity, such as JIT compilation or garbage collection; these timing can be identified and eliminated fairly easily. Another source of potential noise is that the cycle counter available on our platform provides wall clock time, rather than CPU time, so if the operating system switches out the VM process to run another process, the other process's time is included in our method timings. Operating system support to provide thread-specific measurements of CPU time would help reduce these outliers.

The overhead caused by compiling multiple versions of a method for the bakeoff cannot be completely ignored, but can be managed, and the additional overhead is defensible if the technique results in legitimate speedups. As described in Section 6.1, some production VMs already perform additional compilation of hot methods to perform instrumentation for feedback-directed optimization. The overhead introduced by these additional compilations is reduced by performing compilation in the background, and using a spare processor if available. Hardware trends, such as multi-core systems, are likely to help in this regard, especially if the additional parallel cycles provided by the hardware are not fully exploited by the application. These cycles can easily be used by the VM to perform additional optimizations and other runtime services.

Differences in code layout can introduce performance variations, which are not exploited by our current implementation. On many architectures, the position of code in memory can have a significant performance impact. Our current system does not try to place the code for the bakeoff in a good location. In addition, it recompiles each method after the bakeoff completes, which may place the final code in a non-optimal location. However, it is possible to use the performance auditor to help find a good code layout: after the bakeoff completes, the optimized version can be patched to remove the timing code (instead of recompiling). Doing so ensures that any benefits from code positioning identified during the bakeoff are not lost.

Program phase shifts provide both an opportunity and a challenge for any online system. Performing optimizations online allows for the detection of phase shifts, and optimization can be targeted to maximize performance during each phase, which exposes potential performance gains not visible to a less adaptive system. However, a program phase shift can also reduce performance if the system does not re-evaluate optimization decisions when program behavior changes. Online performance auditing, like all feedback-directed optimizations, is susceptible to the potential benefits and degradations due to program phases. Managing the performance risks of performing FDO in the presence of program phases remains an open research area.

Finally, an open question regarding the future use of our work is how to manage the exponential optimization search space. Not only are there dozens of tuning knobs and heuristics that could potentially be applied at runtime, but some of these heuristics have hundreds of potential values; testing all combinations is clearly intractable. However, this exponential space is not created by our work; it already exists, and is essentially ignored by today's systems; a large space does not imply that it is not worth searching, or that substantial performance improvements cannot be attained. Just as there has been a large amount of work exploring the search space with offline empirical search, our framework enables a similar line of online research. We believe that the large search space can be best managed by using a combination of offline *and* online techniques. Extensive offline tuning can be performed to identify the most problematic optimizations and heuristics, and these can become the focus of the online system.

8. Related Work

Prior related work in online empirical evaluations includes work in adaptive compilation and dynamic optimization systems. Prior work in offline empirical search includes work in optimizing libraries, compiler phase orderings, and optimization level usage.

8.1 Adaptive Compilation with Empirical Search

Diniz and Rinard [21] proposed a Dynamic Feedback approach that generates code for several different optimization strategies for important sections of a program. They examine different synchronization strategies for a program's parallel code sections. During execution of these parallel sections, execution alternates between training and production periods for fixed time intervals set in the compiler. The system measures the amount of overhead relative to the performance of the alternative implementations during this training period. It then chooses the implementation with the lowest overhead for the corresponding production period. To choose between implementations, they measure the *overhead* from one execution of the parallel section of the code. By overhead they measure all of the stalls that occur during execution (e.g., stalls due to locking). This approach is only feasible if (a) all of the overhead can be measured, and (b) the amount of stalls seen relative to the overall execution time is independent of the input being run.

Similarly, Voss and Eigenmann [54] perform dynamic optimization on hot spots through empirical search. They use a domain-specific language to specify how to search the optimization space for a specific optimization. As an example, for loop unrolling, a hot spot will be optimized for each level of unrolling. Each of these compiled versions of the hot spot will be run and timed, and the fastest overall time will be kept and used for the hot spot. They time each compiled version only once to decide if it should be used. They deal with varying inputs by partitioning the timings into different bins based on the loop bounds, which relies on loop bound values characterizing varying inputs/workloads.

Both of the above techniques examine the performance of the alternatives only once to choose the better performing one. They argue that performing the timing once for an alternative is sufficient, since the granularity of a single sample can account for a significant amount of execution for the program's they examined. In comparison, for general purpose applications that are run on a JVM it is much harder to create a single large sample that represents the same code being executed. We showed in Section 5 that for our workload that it is difficult to correctly choose which alternative has the best performance with a small number of samples for hot methods in a production Java Virtual Machine. Therefore our approach, based on

statistical analysis and randomization, determines how many samples are needed to make a confident decision.

Fursin et al. [24] explore online empirical search for scientific programs. Prior to the program's execution, a set of optimization strategies are created to be explored during execution. Their system uses phase detection to identify periods of stable, repetitive behavior. During a stable phase of program execution, each optimized version is run once and timed, and the best performing version is chosen. This approach exploits the repetitive behavior of scientific applications. General purpose Java applications have a much higher variance in invocation timings, which motivated our approach of taking a large number of samples.

8.2 Dynamic Optimization Using Heuristics and Feedback Directed Profiling

There is also a large collection of online optimizations that have built-in mechanisms to adapt their strategy based on past program behavior. Such techniques include inlining [19], garbage collection [4], virtual method dispatch [28], object models [8], object layout [34, 44, 30]. and prefetching [12, 45, 14]. The techniques used are specific to each optimization and adapt their strategies based on program behavior rather than execution time. More general purpose systems for continuous optimization [35, 13] have also been proposed.

None of these techniques time and compare multiple optimizations at once to find the best. They instead use profiling information to heuristically guide what to try next, for the specific optimization being examined. In comparison, our goal is to provide an accurate and general method to dynamically measure the exact performance seen when evaluating different optimizations for a hot method.

8.3 Offline Empirical Search

The performance impacts of optimizations can be used to guide performance tuning in libraries and kernels. For example, ATLAS [55], PHiPAC [9], and SPARSITY [32] provide highly-tuned libraries for matrix multiplication and linear algebra kernels, and SPIRAL [43] and FFTW [23] provide digital signal processing solutions. There are even proposals for finding the best-performing sorting routine [38]. STAPL [52] presents a general framework for representing and searching the algorithm space for these types of optimizations.

These techniques work by exploring the algorithm and optimization spaces to tailor the library to the machine it will run on, and even to the application and the inputs the library will be used with. This can result in 50% to an order of magnitude speedups. The best-performing algorithm is found by timing differently compiled versions on test inputs, and this search can take an hour to tens of hours. To reduce the time overheads, recent techniques [57, 11] consider using a model-based approach for searching the optimization space.

Another form of empirical optimization search is offline search of optimization phase orderings and optimization levels. Cooper et al. [17, 16] examined reordering phases with adaptive random sampling and genetic algorithms. These techniques compile a version of the program under different phase orderings and optimization levels, and then time the execution of the resulting program with a representative input to determine which is better.

Kulkarni et al. [37] used genetic algorithms combined with memoization to reduce their search space. They were able to achieve 4% speedups on average by searching the compilation space for 3 hours for embedded applications on an ARM processor. Recently they examined probabilistic pruning of the search space to reduce this search time to 1/3 of their previous approach [36]. Triantafyllis et al. [53] built a decision tree to decide which optimizations to apply, and in what order. The tree guides the search, starting with the most important optimizations. They achieved 5% speedups on average compared to -O2 compilation for the Itanium, while doubling the baseline compilation time.

The above studies have shown that offline search of the optimization space, guided by performance data, can lead to speedups. Recent results [53] show that this search can be done quickly with intelligent decision trees to guide the search. Our work parallels this work: our goal is to achieve similar results with an online search, which explores different optimizations as the program executes.

8.4 Other Techniques for Improving Optimization Decisions

The previous subsection described approaches to improve code quality by performing offline experiments based on measuring execution time. Another approach is to use machine learning to create a model that correlates method properties with effective optimization decisions as measured by execution time [10, 49, 3]. The result is a predictive model that can be used to determine optimization decisions at runtime for any program. This approach has less overhead than performance auditing because performance predictions are generated by simply consulting the predictive model instead of performing multiple compilations and executing the resulting methods; however, its effectiveness is susceptible to the deficiencies of a model as described in Section 2.1.

Another approach is to determine the effectiveness of an optimization based on an evaluation of the quality of the generated code. For example, Dean and Chambers [19] monitor the effectiveness of inlining decisions in the Self system based on how many optimization opportunities are created by subsequent optimization phases. This information is used when considering future decisions to inline a method during the same or subsequent executions. Nethercote et al. [41] also evaluate code quality and use it to possibly reconsider optimization decisions for a method by re-executing earlier optimization phases with different settings to produce better generated code. Compared to performance auditing, this approach also has less runtime overhead; all decisions are made during compilation, no runtime bakeoff occurs. However, because this approach examines the quality of the generated code, it does use an implicit model, and thus, may also be less robust in the presence of varying machine environments.

Our approach, performance auditing, does not use an explicit or implicit model, but instead measures execution time during an online bakeoff to guide optimization decisions.

One can view these four approaches: offline empirical search, offline machine learning, compile-time evaluation, and performance auditing, as complementary approaches on a continuum. As one moves along the continuum from offline empirical search to performance auditing, the cost of the technique increases (such as number of compilations or runtime experiments) as well as the robustness (likelihood for being correct in face of changing inputs and environments). Combining these techniques is both possible and attractive. For example, one could use machine learning or compile-time evaluation to create a small subset of optimization strategies, which could then be explored online by performance auditing.

9. Conclusions

There have been three waves of optimization strategies in virtual machines [5]. The first wave used a JIT compiler on all executed methods, as done in the ParcPlace Smalltalk-80 system [20]. The second wave, pioneered by the Adaptive FORTRAN [26] and Self-93 systems [29], employs selective optimization to focus dynamic compilation resources on the most important methods. The third wave uses profile information gathered during the program's execution to impact optimization decisions.

We believe the fourth wave of optimization strategies will involve virtual machines becoming aware of the performance attained in the current execution environment, and adapting accordingly. Our work is a step in this direction, allowing a virtual machine to determine whether a particular optimization improved, or degraded, bottom-line performance. The technique can be used to tune optimization

decisions at both the macro level (whether to perform an optimization) as well as the micro level (what tuning values to use for an optimization) in a completely automatic online system.

Acknowledgements

The authors would like to thank David Grove for discussions about compiler models, Martin Hirzel for suggesting the title and providing feedback on earlier drafts of this work, and V.T. Rajan for his assistance with the statistical analysis. The authors are also grateful for the feedback from the anonymous reviewers and thank Laureen Treacy for proofreading earlier drafts of this work. This research was funded in part by DARPA contract No. NBCH30390004, and by NSF grants CNS 0311683 and CCF 0541434.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, Feb. 2003.

[2] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. *ACM SIGPLAN Notices*, 39(6):267–276, June 2004. In *Conference on Programming Language Design and Implementation (PLDI)*.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *The International Symposium on Code Generation and Optimization*, 2006.

[4] A. W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, Feb. 1989.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, Oct. 2000. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[6] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation.

[7] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 36(5):168–179, May 2001. In *Conference on Programming Language Design and Implementation (PLDI)*.

[8] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. *ACM SIGPLAN Notices*, 33(5):258–268, May 1998. In *Conference on Programming Language Design and Implementation (PLDI)*.

[9] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *1997 International Conference on Supercomputing*, pages 340–347, 1997.

[10] J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. *ACM SIGPLAN Notices*, 39(6):183–194, June 2004. In *Conference on Programming Language Design and Implementation (PLDI)*.

[11] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *The International Symposium on Code Generation and Optimization*, Mar. 2005.

[12] H. Chen, J. Lu, W.-C. Hsu, and P.-C. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture*, Sept. 2004.

[13] B. Childers, J. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *International Symposium on Parallel and Distributed Processing Symposium*, Apr. 2003.

[14] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. *ACM SIGPLAN Notices*, 37(5):199–209, May 2002. In *Conference on Programming Language Design and Implementation (PLDI)*.

[15] http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.

[16] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

[17] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, May 2002.

[18] http://pag.csail.mit.edu/daikon.

[19] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *LISP and Functional Programming*, pages 273–282, 1994.

[20] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Symposium on Principles of Programming Languages (POPL)*, pages 297–302, Jan. 1984.

[21] P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. *ACM SIGPLAN Notices*, 32(5):71–84, May 1997. In *Conference on Programming Language Design and Implementation (PLDI)*.

[22] M. Frigo. A fast Fourier transform compiler. *ACM SIGPLAN Notices*, 34(5):169–180, May 1999. In *Conference on Programming Language Design and Implementation (PLDI)*.

[23] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *1998 IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[24] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, number 3793 in LNCS, pages 29–46. Springer Verlag, November 2005.

[25] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[26] G. J. Hansen. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. PhD thesis, Carnegie-Mellon University, 1974.

[27] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the pressence of dynamic class loading. In *18th European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 96–122, June 2004.

[28] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *5th European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *LNCS*, pages 21–38, July 1991.

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

[30] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. *ACM SIGPLAN Notices*, 39(10):69–80, Oct. 2004. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[31] E.-J. Im and K. Yelick. Optimizing sparse matrix-vector multiplication for register reuse in SPARSITY. In *International Conference on Computational Science*, May 2001.

[32] E.-J. Im, K. Yelick, and R. Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1), Jan. 2004.

[33] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 38(11):187–204, Nov. 2003.

[34] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):490–505, 2000.

[35] T. P. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[36] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. Exhaustive optimization phase order space exploration. In *The International Symposium on Code Generation and Optimization*, Mar. 2006.

[37] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones. Fast and efficient searches for effective optimization phase sequences. *ACM Transactions on Architecture and Code Optimization*, 2(2):165–198, June 2005.

[38] X. Li, M. J. Garzarán, and D. Padua. Optimizing sorting with genetic algorithms. In *The International Symposium on Code Generation and Optimization*, pages 99–110, Mar. 2005.

[39] D. Maier, P. Ramarao, M. Stoodley, and V. Sundaresan. Experiences with multithreading and dynamic class loading in a Java just-in-time compiler. In *The International Symposium on Code Generation and Optimization*, Mar. 2006.

[40] P. Nagpurkar, M. Hind, C. Krintz, P. F. Sweeney, and V. Rajan. Online phase detection algorithms. In *The International Symposium on Code Generation and Optimization*, Mar. 2006.

[41] N. Nethercote, D. Burger, and K. S. McKinley. Self-evaluating compilation applied to loop unrolling. Technical Report TR-06-12, The University of Texas at Austin, Feb. 2006.

[42] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 1–12, Apr. 2001.

[43] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation.

[44] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions on Embedded Computing Systems*, 2(2):1–32, May 2003

[45] R. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Conference on Parallel Architectures and Compilation Techniques*, 1996.

[46] http://www.sable.mcgill.ca/software/#soot.

[47] Standard Performance Evaluation Corporation. SPECjbb2000 Java Business Benchmark. http://www.spec.org/jbb2000.

[48] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

[49] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *The International Symposium on Code Generation and Optimization*, pages 123–134, 2005.

[50] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–195, Nov. 2001. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[51] The DaCapo Project. DaCapo Benchmark Suite, version beta051009. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.

[52] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288, 2005.

[53] S. Triantafyllis, M. Vachharajani, and D. August. Compiler optimization-space exploration. *Journal of Instruction-Level Parallelism*, 7:1–25, Jan. 2005.

[54] M. J. Voss and R. Eigemann. High-level adaptive program optimization with ADAPT. *ACM SIGPLAN Notices*, 36(7):93–102, July 2001.

[55] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[56] http://xml.apache.org/xerces2-j/index.html.

[57] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.