Improving Virtual Machine Performance Using a Cross-Run Profile Repository

Matthew Arnold[†] Adam Welc[‡†] V.T. Rajan[†]

[†]IBM T.J. Watson Research Center
fmarnold,vtrajang@us.ibm.com

[‡]Purdue University
welc@cs.purdue.edu

ABSTRACT

Virtual machines for languages such as the Java programming language make extensive use of online profiling and dynamic optimization to improve program performance. But despite the important role that profiling plays in achieving high performance, current virtual machines discard a program's profile data at the end of execution, wasting the opportunity to use past knowledge to improve future performance.

In this paper, we present a fully automated architecture for exploiting cross-run profile data in virtual machines. Our work addresses a number of challenges that previously limited the practicality of such an approach.

We apply this architecture to address the problem of *selective optimization*, and describe our implementation in IBM's J9 Java virtual machine. Our results demonstrate substantial performance improvements on a broad suite of Java programs, with the average performance ranging from 8.8% – 16.6% depending on the execution scenario.

Categories and Subject Descriptors

D.3.4 Processors [**Programming Languages**]: [Optimization]

General Terms

Languages, Measurement, Performance

Keywords

Java, virtual machine, profiling, selective optimization

1. INTRODUCTION

Virtual machine technology has progressed significantly over the last several years, largely due to the success of the Java™ programming language, and more recently the C# language. The dynamic nature of these languages has spurred particular interest in the area of dynamic compilation and adaptive optimization [1, 2, 5, 11, 14, 21]. Most of the production Java Virtual Machines available today contain advanced adaptive optimization systems that monitor and optimize the program as it executes, and these systems have a substantial impact on performance.

These systems use profiling information in multiple ways to improve performance. First, they identify the frequently executed parts of the program to determine where optimization efforts should be focused. This process, referred to as *selective optimization*, has been shown to be effective at keeping compilation overhead to a minimum while producing efficiently executing code [1, 12, 21]. The second use of online profiling is to perform feedback-directed optimization, where profiling information is used to improve the quality of generated code [3, 11, 18, 21], giving them the potential to outperform a static compilation model.

Despite the value of the dynamic information collected, it is usually discarded at the end of the execution [1, 5, 11, 12, 14, 21]; the VM learns nothing from previous runs, and profiling is started from scratch when the program is executed again. Capitalizing on this wasted opportunity seems like an obvious next step in the evolution of adaptive optimization systems. Exploiting *offline* profile data is certainly not a new idea, and has existed for many years [18]. However, technical challenges prevent the straightforward application of these techniques in a fully automatic virtual machine. For example, traditional offline profiling assumes a clear distinction between *training* runs, where profile data collected, and *production* runs, where profile data is exploited. A VM does not have the luxury of this distinction; every run is both a production run *and* a training run, thus the VM must be prepared to adapt to continuously changing profile data. In addition, the VM now has two potential sources of profile data: offline and online. The offline information is potentially more complete, containing data from multiple completed runs, yet the online information is possibly more relevant because it is specific to the *current* run.

For long-running applications there is less motivation to store information across runs, because their running time is long relative to the time it takes the adaptive system to characterize their performance. But for short and medium length applications, the VM will likely exit before it was able to fully exploit the profile data it collected, and will continue to miss the same opportunities each time the application is executed. In fact, there are many realistic scenarios where this occurs. For example, in an edit-compile-debug cycle, a program like the `javac` bytecode compiler may be executed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
Copyright 2005 ACM 1-59593-031-0/05/0010 ...$5.00.

repeatedly on the same set of (slightly changing) input files. Similarly, when developing and debugging a J2EE server application, it is sometimes necessary to shutdown and restart the application server frequently. In both of these cases, opportunity is lost while the VM starts learning from scratch at the beginning of each run.

In this paper, we describe an architecture for augmenting a virtual machine with a *profile repository*, allowing the VM to remember profile data and optimization decisions across program executions. Our system is fully automatic; there is no distinction between training and production runs, and the user does not need to know that the repository exists. Central to our approach is the ability to use a combination of both online and offline profiling information together when making optimization decisions at runtime.

The contributions of this paper include:

- We present an architecture for augmenting a VM with a profile repository, allowing profile data to persist across runs of an application. This architecture addresses many of the the challenges that previously limited the practicality of such an approach.
- We show how this architecture can be applied to the specific problem of *selective optimization*.
- We describe our implementation and empirical evaluation using IBM's product J9 JVM. Our results demonstrate substantial performance improvements on a broad suite of Java programs.

The rest of the paper is organized as follows. Section 2 presents our general architecture and Section 3 describes how it can be applied to selective optimization. Sections 4 and 5 describe our implementation and experimental results respectively. Section 6 presents related work and Section 8 presents our conclusions.

2. SYSTEM ARCHITECTURE

Our architecture provides the virtual machine with the ability to write to and read from a persistent repository, thus giving it the ability to remember information across application runs. This approach has a number of nice characteristics.

- It enables the VM to exploit profiling information earlier in the program's execution, and enables feedback-directed optimization over the life of the application, rather than just a single execution.
- It is fully automatic, and requires no explicit training.
- The only demand on the VM's environment is some extra temporary disk space; it can be cleared or taken away at any time, similar to a cache.
- The total disk space required is modest because information is recorded only for frequently executed parts of the program.
- Writing to and reading from the repository is optional; if the repository becomes corrupted for some reason, it can simply be ignored.
- No new security concerns are introduced because the repository does not affect program correctness.

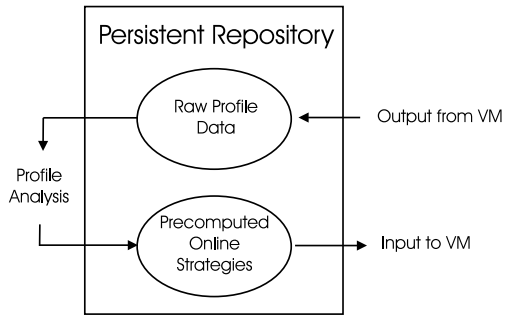

Figure 1: Profile repository architecture

However, this approach does present a number of fundamental challenges, such as:

1. How should the VM make decisions from aggregate profile data that is continually changing? The adaptive system cannot scrutinize a full history of program executions each time it needs to make an optimization decision. Furthermore, the profile data may contradict itself, with different executions demonstrating different behavior.
2. How should the offline information in the persistent repository be used together with the online information that is collected at runtime? The offline repository provides summary of previous runs, but the online information provides up-to-date details about the *current* execution.

In addition, there are infrastructure challenges that any production implementation would need to address: reading and writing to the repository must be efficient, otherwise the time spent accessing the data could outweigh the benefit. The repository must also be reasonably robust to allow simultaneous access by multiple VM instances without becoming corrupted.

In the next subsections, we present our general architecture for the repository and describe how it can be used to address the above challenges.

2.1 Repository Architecture

Figure 1 presents our architecture for the profile repository. It contains two separate components. The first component, labeled *raw profile data*, is where the profile data is aggregated and stored. Examples of possible raw profile data include time spent in the various methods of the program, dominant call targets at virtual call sites, and intra-procedural control flow paths. Each time a VM contributes back to the repository, its profile information is merged together with the existing raw data in the repository.

The second component of the repository is labeled *precomputed online strategies*. These strategies are short and concise suggestions to the adaptive optimization system regarding the actions it should take at runtime. Examples of online strategies include instructing the VM to optimize a method earlier or later than usual, or to inline a particular call target at a virtual call site. These online strategies can be read at VM startup, allowing the VM to exploit the benefits of the persistent repository without needing to analyze, or even read, any of the raw profile data. Precomputed online strategies are discussed further in Section 2.3.

The goal of the *profile analysis* component in Figure 1 is to decide how optimization should be performed based on the raw data in the profile repository. The analysis can be performed at any time, such as by an offline agent that runs in the background. Our system performs this analysis at runtime, while the JVM is shutting down, as described later in Section 2.4.

None of the actions associated with the arrows in Figure 1 is required to be performed during an individual VM instantiation. A VM can read and use the precomputed strategies without later contributing back to the raw profile database; similarly, it can contribute back to the raw profile info without having read the precomputed strategies, or performing the profile analysis.

2.2 Repository organization and association

Most virtual machines perform optimization at a method granularity, so it is natural to store profiling data at a method granularity as well. Previous work has suggested attaching profiling information, or optimization directives, to the Java bytecode as annotations [13]. However, we chose not to modify the program source in any way because it would violate our goal of being transparent to the user and could risk corrupting the program source. Instead, we maintain a separate repository in temporary disk space reserved for the virtual machine.

To maintain this repository, the VM needs the ability to map information in the repository to the program being executed at runtime. This mapping is not entirely trivial because multiple programs may have classes or methods with the same name and signature. In addition, some methods (such as library methods) are shared by multiple programs. In our design, we keep a separate entry in the repository for each *program*, where a program is defined by the fully qualified signature of the `main()` method that is executed to start program execution; the location of the classfile on disk can be added to avoid merging multiple programs whose `main()` method share the same fully qualified class name.

For each program in the repository, information is recorded for a subset of the methods in the program, namely the *hot* methods that are executed frequently. If a particular method is executed by multiple programs, the method is recorded separately for each program (assuming it was considered *hot* in both), allowing the analysis of the method to be specialized in the context of each program.

2.3 Precomputed Online Strategies

In our architecture, the output of the profile analysis is a *precomputed online strategy*. This precomputed strategy is not a fixed optimization plan, such as "optimize method M upon first invocation." Instead, it is a list of *conditional* optimization actions that tell the online system system what to do based on what happens at runtime. For example:

```
if (runtime condition 1)
  perform(optimization action 1);
if (runtime condition 2)
  perform(optimization action 2);  ...
```

This general structure provides power and flexibility regarding the solutions that can be constructed by the profile analysis phase, allowing the analysis to construct plans that exploit offline aggregate profile data *together* with the online profile information collected during the current execution.

If a particular aspect of a profile was identical in *all* previously observed executions of a program, conditional optimization directives may not be needed; the optimization could be applied unconditionally. For example, if a virtual call site has the same dominant call target in all executions, the analysis may decide to inline the method unconditionally.

However, it is possible that the repository contains profile data that contradicts itself for different runs of the program, such as a call site that is dominated by calls to method A in some executions, and method B in others. In this case, a conditional online strategy makes it possible to instruct the online system to look for certain conditions before applying a particular optimization. The online system could first determine whether the execution is biased toward method A or B, then invoke the appropriate optimization plan.

2.4 Profile Analysis

The goal of the *profile analysis* component is to construct an online strategy based on the profile data in the repository. This analysis is not required to be performed when the repository is being read or written, thus it could potentially be performed by a separate offline analysis engine, or possibly using a background process that runs when the processor is idle.

However, many users prefer not to have uninvited background processes running on their system; therefore, we have developed our system to perform the profile analysis at the time the profile repository is updated by the VM. When the program exits and the VM is shutting down, its profile data is merged into the profile repository, and the profile analysis is performed to update the precomputed online strategies. This computation is performed while the VM is active, so the amount of work performed must be limited or it may be perceived as overhead by the user.

Constraining the overhead limits the types of analysis that can be performed; however, only a subset of the (hot) methods in the program need to be recorded and monitored, reducing the amount of work to be performed. If the analysis is still too expensive, there are a number of options. First, it is not necessary to compute the online strategies from scratch each time the repository is updated. Online strategies can be adjusted incrementally to account for the new information recently contributed to the repository. If the profile analysis is designed as an iterative solution procedure, the previous online strategy can be used as the initial solution to the algorithm. A small number of iterations are likely to be needed, given that the change to the profile repository after each execution will be minimal. To limit overhead, the number of iterations per VM instantiation could be limited to some fixed number (possibly even 1), so that the work is distributed over multiple executions of the program; the online strategies will become more refined as the number of program executions increases.

Furthermore, if overhead is still a concern, the VM can exploit the fact that the online strategies do not need to be fully updated after every execution of the VM. The VM can impose a limit on the amount of time it spends performing the profile analysis; by remembering the point at which it left off, it can resume the next time the program executes. The time limit could be a function of the VM execution time, to ensure that total overhead remains low.

2.5 Profile repository decay

To rid the repository of old, stale information, the data in the repository can be periodically decayed, similar to the profile decay mechanism used online by the Self-93 system [12]. Using decay has the advantage that it gives more precedence to recent runs, and prevents the repository from growing indefinitely.

3. SELECTIVE OPTIMIZATION

This section describes how we instantiate our general architecture to address the problem of *selective optimization*. Selective optimization is the process of determining what parts of the program should be optimized, when optimization should be performed, and what level of optimization should be used. In most virtual machines, selective optimization is likely to be the most important determining factor of VM performance for short to medium length executing programs. Dynamic compilers have become quite complex; high optimization levels produce good quality code, but at the cost of increased compilation time; using them judiciously is critical for achieving good performance.

Traditional selective optimization uses online profiling to predict which methods will run long enough to justify being optimized at a higher level. Many virtual machines use simple counter schemes, triggering compilation when a method's invocation count (and possibly loop backedge count) exceeds a fixed threshold [5, 11, 14, 21]. Other systems [1] use a more formal cost-benefit model for making online decisions. Our system also uses this cost-benefit approach, as discussed later in Section 3.3.1.

The goal of our system is to use the profile repository to make better predictions about a method's future running time. In the sections that follow we describe the details of our repository when used for selective optimization. Specifically, we discuss what information is stored in the profile repository, the structure and use of the precomputed online strategies, and our algorithm for constructing the online strategies.

3.1 Profile repository: histograms

If every program had only one input, predicting a method's total future running time (during the execution of that program input) would be simple. Virtual machines already profile the application and maintain a rough breakdown of the amount of time spent in the hot methods [1, 11, 21]; this information could be written to the profile repository, and fed into the program the next time it executes.

However, most programs have multiple inputs, and these inputs can drastically affect the running time of the program, as well as the distribution of time spent in the various methods of the program. Therefore, for each method in our profile repository we maintain a *histogram* of running times, providing a *probability distribution* of the ending time of each method. Given a method M, and a time T, the histogram for M tells us the number of program executions for which the M executed for T time units before exiting.

Any unit of time, such as cycle count or wall-clock time, could be used for recording time spent in a method. Our virtual machine profiles time spent in methods using timer-based sampling, thus *method samples* is the unit of time used throughout our system.

When the VM exits, the information from the current run is added to the aggregate information in the repository. For each method that was sampled during the execution, the corresponding histogram is retrieved from the repository (or created if it does not exist) and the correct histogram bucket is incremented, depending on how long the method ran in the current execution. For each method in the repository that was not sampled during the current execution, the bucket for time zero is incremented. Note that this solution tracks only methods that were sampled at some point during the execution.

To minimize repository space and I/O time, we may map multiple times values to the same histogram bucket as the value of T increases. Distinguishing the histogram values for consecutive times T_i and T_{i+1} is important for small values of i, but becomes less important when i becomes large. For example, distinguishing between 1 vs 5 samples is important; distinguishing between 501 and 505 samples is not. Therefore, we use a non-uniform bucket size in our histogram. The first N buckets each correspond to a single time unit; after time N the bucket size increases polynomially to a maximum number of buckets. All samples that occur beyond the maximum bucket are recorded using the last bucket.

3.1.1 Accounting for the effects of optimization

As optimization occurs, the distribution of time spent in the various methods of the program changes. Optimizing a method, M, reduces the amount of time it spends executing. A system using a naive approach to profile optimized code will observe a reduced runtime for method M, and may conclude that it no longer requires such a high level optimization. This effect can lead to poor optimization choices, and oscillation in the optimization decisions over time.

Therefore, our system accounts for these effects by recording all method times in *unoptimized time units*, which represent the amount of time the method would have executed if it had *not* been optimized. Since our system measures time using method samples, our time unit becomes *unoptimized-samples* (the number of samples the method would have received if it had not been optimized).

To profile the application using *unoptimized-samples*, the samples are scaled as they occur at runtime. When an unoptimized method is sampled, the sample count is incremented by 1 unit; when an optimized method (optimized at level j) is sampled, the sample count is incremented by the relative speedup between optimization level j and unoptimized code. For example, assume that code optimized at level j executes roughly 3 times faster than unoptimized code; when a method compiled at level j is sampled, the sample count is incremented by 3 units, rather than 1 unit. The resulting sample count is an approximation of the sample count that would have occurred if the method had not been optimized.

This methodology allows profiles from multiple runs to be stored in a uniform fashion, regardless of what optimization occurred at runtime.

3.2 Precomputed online strategies

A precomputed strategy for selective optimization instructs the online system when to compile methods, and at what optimization level. Each precomputed strategy consists of a set of tuples, $\langle time, optLevel \rangle$ and each tuple corresponds to a method being compiled by the optimizing compiler; *time* is the amount of time the method needs to

execute (in our case the number of samples needed), and *optLevel* is the optimization level to be used. For example, the strategy $\{\langle 1,2\rangle, \langle 3,4\rangle\}$ directs the adaptive system to compile at optimization level 2 after the first sample, and at level 4 after the third sample.

Using these precomputed strategies at runtime is relatively straightforward. When a method M is sampled, the system checks whether there is a precomputed online strategy for M. If so, it uses the current number of samples for M, together with the precomputed strategy, to determine what, if any, optimization should be performed. Following a precomputed strategy is actually simpler, and more efficient, than the online model-based cost/benefit analysis performed by our system when precomputed strategies are not available.

If a precomputed strategy does not exist for the sampled method, or if the number of samples is beyond the maximum number of samples previously observed for this method, the system simply falls back to the default online recompilation behavior.

3.2.1 Preserving the benefits of delayed compilation

If a method is to be optimized at some point during an execution, performing that optimization earlier is generally more beneficial because it will maximize the amount of execution that occurs in the optimized version of the code.

However there are some *disadvantages* to performing optimization too early. When compilation is delayed, the optimizer has more information about the program available, such as knowing more about the program's class hierarchy, thus providing more opportunities for speculative inlining. Other examples include having more information about the sizes of types, allowing more efficient inlined allocation sequences. If the method is compiled before execution begins, these advantages are lost.

Our system uses a simple solution to perform compilation as early as possible *without* giving up the advantages of delayed compilation. If the precomputed strategy specifies that method M should be optimized at time 0 (before the method begins executing), the optimization is not actually performed until the *second* invocation of method M. The first invocation of M executes unoptimized, giving the VM time to see code before it is optimized, and gain many of the benefits of delayed compilation.

However, some programs contain long-running methods that are invoked a small number of times, possibly only once. Failing to optimize these methods prior to their first execution means that they may be stuck executing in the unoptimized version indefinitely if the system does not perform on-stack replacement [10]. Our system identifies these methods by observing the large number of unoptimized samples that occur, and compiles them prior to their *first* invocation in the next execution.

3.3 Model-based selection of online strategies

Given a method's execution time probability distribution (histogram), the profile analysis needs to construct an online strategy. We initially investigated a number of simpler, heuristic-based algorithms, but abandoned this approach quickly after seeing how poorly they performed in all but the most simplistic scenarios. Instead, we developed a purely model-based algorithm for constructing the online strategies.

3.3.1 Modeling the effects of compilation

The version of the J9 JVM used in this work makes online optimization decisions using a *cost-benefit* model, similar to that used in Jikes RVM [1]. The same performance model can be used for constructing a optimization strategy based on profile data in the repository.

When compiling a method M at optimization level j, the model can be used to estimate two quantities:

1. **cost(M,j):** the amount of compile time that will be consumed if method M is optimized at level j.
2. **speedup(M,j):** the performance gain of executing method M optimized at level j, relative to unoptimized code.

Our system estimates these quantities based on offline performance measurements. The compile time cost is computed as a linear function of method size, based on the *compilation rate* (bytecodes/millisecond) of the compiler at level j. The speedup of each optimization level is a constant time-reduction factor relative to unoptimized code.

3.3.2 Objective function

The goal of our algorithm is to construct an online strategy R that maximizes some characteristic of overall performance. The choice of objective function may vary depending on the desired performance goals for the system. For a general purpose virtual machine, we chose an object function that will maximize *average performance* if the history in the profile repository were to repeat itself. More formally, for a given method M, let $r_0, r_1 ... r_n$ represent the individual runs of method M recorded in the profile repository. Our algorithm selects a strategy R that minimizes:

$$\sum_{i=1..n} \frac{R(r_i)}{unopt(r_i)} \quad (1)$$

where $R(r_i)$ and $unopt(r_i)$ represent the running time of the r_i when executed using strategy R, and when executed unoptimized, respectively.

Note that this optimization function is different than minimizing total, or average *running time*, which would give more weight to longer running executions of a program. For example, assume that method M runs for 1 second in r_1 and 100 seconds in r_2, and there is a fixed optimization strategy that performs 1 second of compilation at VM startup; this optimization strategy may minimize the sum of the two executions, but would double the running time of r_1. By evaluating the performance relative to unoptimized code, our approach gives equal weight to all executions of the program recorded in the repository, independent of their running times.

3.3.3 Algorithm: online strategy selection

Our algorithm works on a single method at a time, and uses a dynamic programming approach to compute a strategy that minimizes the objective function above for a method M. The running time of the algorithm is $O(N * K^2)$ where N is the number of buckets in method M's profile distribution (histogram), and K is number of optimization levels. K is expected to be a small constant (in our system $K = 4$), thus the complexity is linear in the size of the histogram.

An intuitive description of the algorithm is as follows. The algorithm begins at the end of time and walks backward. For the current point in time t, the algorithm asks the following question for each optimization level j: "If method M is currently optimized at level j, what is the optimal strategy to take from time t forward?" The optimal solution has already been computed for time $t+1$ (for all optimization levels), thus the algorithm needs only consider the effects from time t to time $t+1$.

The histogram of method ending times is used to determine the number of program runs in which method M executes for at least t time units; performing compilation at time t adds cost (and benefit) to only those runs.

When considering whether to optimize M at a higher optimization level h at time t, the algorithm considers three factors:

1. The cost of performing the compilation at time t. This cost affects only runs in which method M executed for at least t time units.

2. The benefit of executing for one time unit (from time t to time $t+1$) at level h, rather than level j. The algorithm credits this benefit only for runs where M executed for at least $t+1$ time units.[1]

3. The benefit of being optimized at level h from time $t+1$ forward. This value was already computed in the previous iteration of the algorithm.

If moving from level j to level h at time t is better than staying at level j, then this compilation is recorded as part of the optimal strategy. The algorithm continues moving backward through time until time 0 is reached. The final strategy reported is the optimal strategy computed for starting at time zero at optimization level 0 (unoptimized).

The formal description of the algorithm is presented in Figure 2. Let $runsExecutingM(t)$ represent the number of programs runs that execute method M for t time units or more (computed from the profile histogram). Let $j = 0...K$ represent the optimization levels of the compiler, where level 0 represents unoptimized code. Let C_j represent the compile time cost of M at optimization level j, and let S_j represent the speedup of optimization level j relative to unoptimized code ($S_j = 0.5$ if optimization level j is twice as fast as unoptimized code). Variable F_j represents the optimal cost of executing the program from time $t+1$ forward, assuming method M was already optimized at level j; $Strat_j$ represents the strategy that achieves time F_j.

3.3.4 Bounding compilation for unexpected inputs

The above algorithm will maximize average performance *only if* future executions of the program occur as predicted by the profile repository. If a new input demonstrates radically different behavior from previous runs, the performance could be arbitrarily bad relative to the original system. For example, if method M is predicted to be long-running, our algorithm may select a strategy that optimizes M at a high level of optimization at time zero. This time spent on compilation may lead to poor performance (relative to the original system) if a future input causes M to run for a short amount of time.

To ensure reasonable performance for unpredicted program inputs, we parameterize the algorithm with a *compilation bound*. Given a compilation bound of $X\%$ the algorithm discards solutions that would increase compilation time by more than $X\%$ relative to the original system. A small constant ϵ is added to running times to enable calculations at time zero. Without this smoothing factor, no compilation would ever be performed at time zero for any performance bound.

To construct precomputed strategies that meet the requirements of the compilation bound, the inner loop of the algorithm is modified as follows. Let $BOUND$ be the compilation bound and ϵ be the smoothing factor described above.

```
  for each optimization level h such that h > j
      origComp = max compilation time that
                 could be performed by original
                 online system at time t
      budget = (origComp + ε) * (1 + BOUND)
      if (C_h > budget)
         // Skip C_h.  Too expensive at time t
         continue;
      [remainder of original inner loop]
```

```
Input: C_j, S_j, runsExecutingM(t)

F_j = Strat_j = 0 for all j // No future running time

iterate backward in time, t = t_max .. 0

  for each optimization level j = K .. 1

    // Optimal future time from time t forward
    // if no compilation performed at time t.
    // Divide exe time by t to scales for
    // average performance (see Section 3.3.2)
    exeTimeThisUnit = (S_j * runsExecutingM(t+1))
    minCost = F_j + exeTimeThisUnit/t
    action = {}   // How to achieve minCost

    for each optimization level h such that h > j
       // Optimal running time if M is compiled at h
       // at time t.  Divide exe time by t to scale
       // for average performance
       compilationTime = C_h * runsExecutingM(t)
       exeTimeThisUnit = S_h * runsExecutingM(t+1)
       cost= F_h + (compilationTime + exeTimeThisUnit)/t
       if cost < minCost
          minCost = cost
          action = compile from level j to level h
                   at time t_i

    // We now have the optimal strategy for time t
    // forward when starting at level j
    Fj = minCost
    Strat_j = Strat_j ∪ action

Output: F_0, Strat_0
```

Figure 2: Algorithm for computing a precomputed online strategy from a method's probability distribution (histogram).

[1]Method M's execution time may be between time t and time $t+1$, but due to the profile granularity it can not be known precisely. We make the assumption that optimization performed at time t does not benefit these executions. Alternatively, one could assume that on average the method executed for time $t+0.5$.

3.4 Warming up the repository

One question that arises is how many executions should be observed before the information in the repository should be trusted and used for optimization. Should aggressive optimization decisions be made after observing a single execution? Should an arbitrary, fixed number of runs be required?

Our solution is to exploit the compilation bound (described above, in Section 3.3.4) to slowly increase the aggressiveness of the decisions made based on the repository. The compilation bound begins at 0.0%, which allows no additional compilation to be performed relative to the underlying online model-based system, thus effectively leaves the original system unchanged.

Over the first N executions, the compilation bound is increased linearly until it reaches some maximum value. As additional program executions occur and the bound increases, the optimization decisions will become more aggressive until the maximum value of the bound is reached.

The values our system uses for these variables is described in Section 4.

4. IMPLEMENTATION

We implemented our technique in the J9 Java Virtual Machine [11], one of IBM's production Java virtual machines. J9 is a fully functioning, high-performance JVM that contains an interpreter, a JIT compiler with multiple optimization levels, and an adaptive optimization system. The performance of the J9 JVM is competitive with that of other leading production JVMs.

The version of J9 used for this work has been extended to use a cost-benefit model for making online optimization decisions, similar to that used by Jikes RVM [1]. This model-based version offers highly competitive startup performance and is used as the base for all of our experiments in this paper.[2]

Our initial implementation of the repository is stored on disk as a simple text file, and methods are written based on their signature (class name, method name, and arguments). Our current implementation does not distinguish methods with the same signature that are loaded with different class-loaders. Similarly, our current system does not automatically extract the profile data for the executing program from the repository; the profile data for each program is manually specified by the experimental test harness. These limitations are not fundamental to the ideas being explored, and could be addressed in a production implementation of our system.

5. EXPERIMENTAL EVALUATION

This section presents an empirical evaluation of our work.

5.1 Benchmarks

The first column of Table 1 lists the benchmarks used in this study. The first seven benchmarks are the SPECjvm98 benchmark suite [20]. `Pseudojbb` is a modified version of `SPECjbb2000` that executes a fixed number of transactions, rather than executing for a fixed time period. `Ipsixql` is a benchmark of persistent XML database services [7]. The `xerces` benchmark measures a simple XML Parser exercise [23]. `Daikon` is a dynamic invariant detector from MIT [8]. `Soot` is a Java bytecode analysis framework from McGill University [19]. `Saber` is a J2EE code validation tool developed within IBM [15]. `Cloudscape` is an open source relational database system. [6]. `Eclipse` is the Eclipse Platform IDE [9]; to benchmark Eclipse, we measure the time it takes to start Eclipse and initialize a workspace given as input.

[2]The version of J9 used for this study is an experimentally modified version, thus the performance timings reported do not represent official performance results for IBM's J9 Java Virtual Machine. In addition, this experimental version does not constitute a product statement about any future releases of the product JVM.

Program	Small Input Time (s)	Small Input Meth exe	Small Input Size (KB)	Large Input Time (s)	Large Input Meth exe	Large Input Size (KB)
db	0.38	901	841	10.32	902	841
jess	0.46	1281	1009	2.92	1295	1011
mtrt	0.60	1091	870	2.49	1073	870
jack	0.77	1090	940	3.69	1090	940
javac	0.81	1617	1147	5.44	1647	1147
mpegaudio	0.83	1029	931	3.55	1027	931
compress	1.14	891	839	8.92	890	839
ipsixql	0.62	983	785	3.91	1013	786
xerces	1.04	1513	1129	2.61	1512	1129
pseudojbb	7.46	1448	938	25.16	1447	940
soot	1.43	2136	1648	29.58	2768	1736
saber	2.72	3533	2738	17.02	4649	3097
daikon	3.01	2748	1839	16.04	1497	1840
cloudscape	2.01	5669	4026	18.28	9234	5961
eclipse	6.25	10119	7089	18.67	18767	14713

Table 1: Benchmarks used in this study.

To evaluate startup performance, programs were benchmarked by recording the *total execution time*. Benchmarks that contain an iterative driver, such as the SPECjvm98 benchmark suite, were configured to run only a single iteration of the benchark. Time spent reading, writing, and processing the persistent repository is included in the timings.

For this entire paper, a "run" of a program always consists of starting a new JVM and running the program to completion on a given input. This should not be confused with benchmarking methodologies that use a harness to execute multiple iterations of the program within a single JVM instance, such as the official SPECjvm98 reporting procedure.

To produce a wide range of running times for the benchmark suite, each benchmark is run with two different sized inputs: "small" and "large". For each input size, Table 1 reports the following three quantities: 1) running time of the benchmark (seconds) when run with the base version of J9, 2) the total number of methods executed, and 3) the total size of the program in kilobytes, measured as the sum of the sizes of the classfiles loaded by the virtual machine at runtime. The running time of these benchmarks ranges from 0.4–29.6 seconds.

All numbers reported in this section are the median of 10 runs to reduce noise. The experiments were performed using Red Hat Linux version 9.0 running on an IBM Intellistation with a 3.0GHz Intel Xeon processor and 1 gigabyte of RAM.

5.2 Methodology

The performance of our system depends heavily on the usage scenario of the virtual machine. Specifically, performance depends on a) the inputs used to train the profile repository, and b) the inputs used during the performance

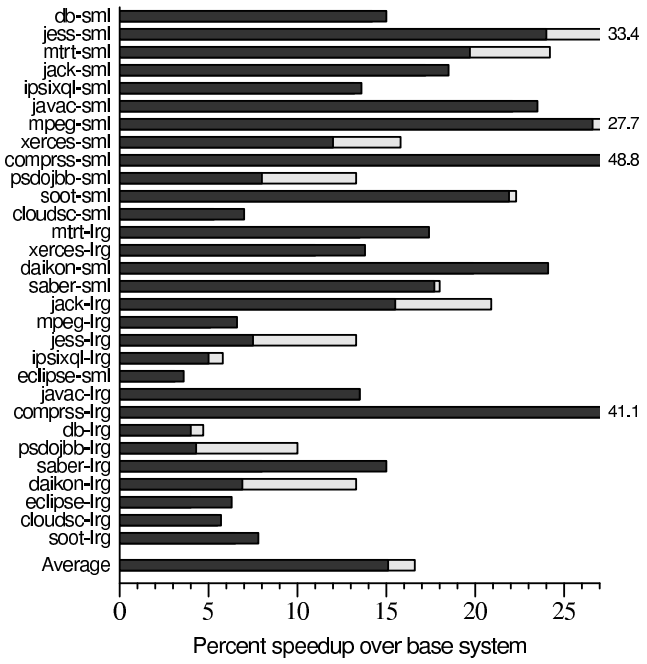

Figure 3: Single input scenario, steady-state performance (performance when repository is warm).

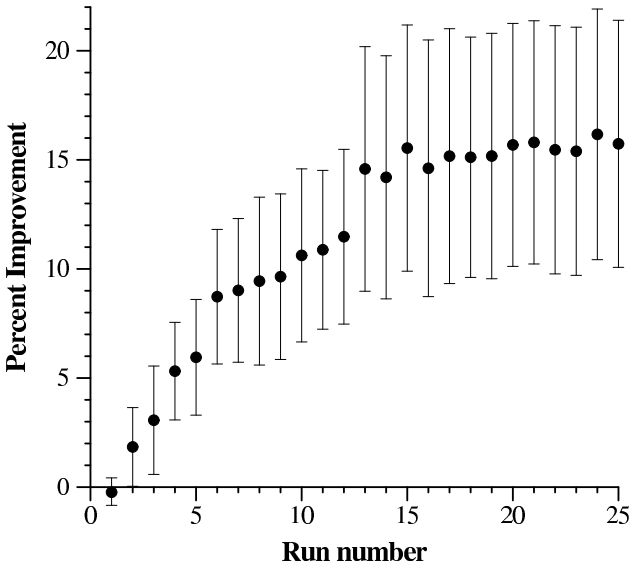

Figure 4: Single input scenario, behavior over time as repository warms up. Performance averaged over all benchmarks.

evaluation. To provide a thorough understanding of the performance implications of our technique, we evaluate its performance using the following four usage scenarios.

- **Single input**: program is executed 25 times with same input. This scenario presents the best case scenario for our technique.
- **Dual input**: program is executed 25 times, alternating between the small and large inputs. This scenario demonstrates the ability of our technique to be effective in the presence of bimodal program running times.
- **Phase shift**: program is executed 10 times with one input, followed by 10 executions using the second input. This scenario shows the behavior when there is a change in program usage.
- **Multi input**: an expanded range of inputs is used (10 for each program). The program is then executed 50 times using all of the 10 inputs in a randomly selected order. This scenario shows the behavior of our system when used over a wide range of inputs.

For each scenario, performance is characterized in two ways, `warmup` and `steady-state`. `Warmup` shows the behavior of the system over time while the profile repository is warming up (as described in Section 3.4). `Steady-state` shows the performance that is eventually converged upon if the input sequence is repeated long enough for the repository to become fully warm.

Sections 5.3–5.6 describe each of the 4 user scenarios, respectively, and Section 5.7 provides miscellaneous statistics about our system.

5.3 Single input scenario

This scenario evaluates the performance of our technique when the program is run repeatedly with the same input, thus representing a best case scenario for our technique. This scenario should not be regarded as a theoretical limit study, however, because this performance would be achieved in practice if a user executes a program repeatedly with the same input, or inputs, that result in similar behavior.

Figure 3 presents the `steady-state` performance results. Each program and input pair is represented by a bar in the graph. The x-axis represents the performance improvement of our system relative to the base system (described in Section 4), which uses only online information for making compilation decisions.

The full length of the bar shows the performance of our system with no *compilation bound* (see Section 3.3.4), thus arbitrarily aggressive compilation decisions can be made at any time during execution. In this scenario, our system offers tremendous speedups over the base system, ranging from 3.6% to 48.8% improvement, with an average of 16.6%.

The solid black portion of the bar represents the performance of our system with a compilation bound of 25%. Enforcing the compilation bound reduces the speedups for some of the benchmarks, such as `pseudojbb-small`, `jess-large` and `daikon-large`, but overall the performance was similar to the unrestricted case ranging from 3.6% to 47.2% with a geometric mean of 13.9%. The 25% compilation bound offers most of the benefit of the unrestricted case, and provides more robust performance for unseen inputs, thus is the configuration we use for the remaining experiments.

Recall that these speedups are relative to an already high-performing system, as described in Section 4; it is our best tuned version of a model-based selective optimization system, and is implemented in a production JVM. The large speedups obtained demonstrate the performance potential of selective optimization. For short and medium length applications, improving the *policy* for making compilation de-

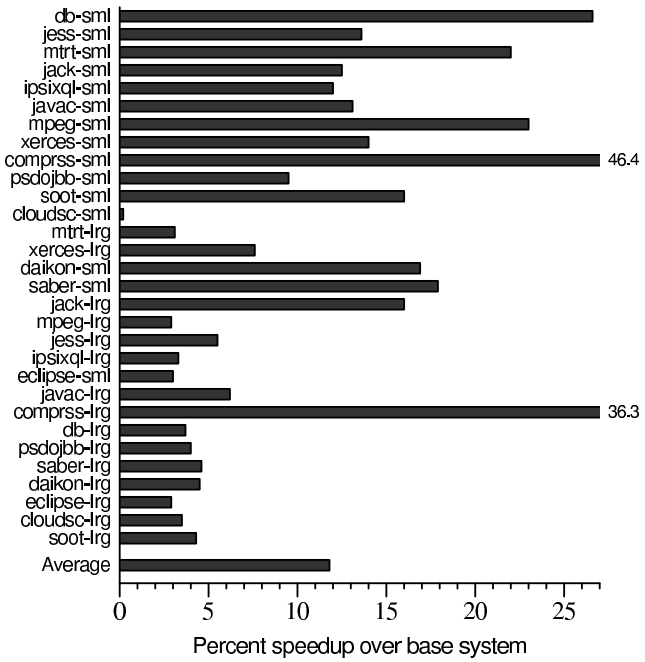

Figure 5: Dual input scenario, steady-state performance.

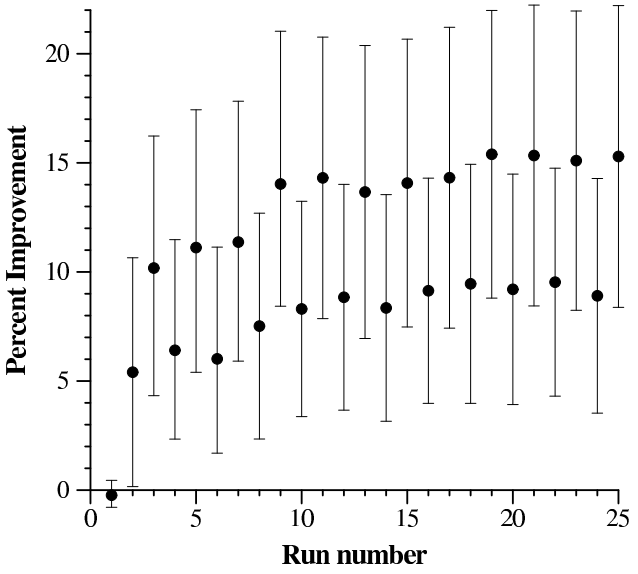

Figure 6: Dual input scenario, behavior over time. Performance averaged over all benchmarks.

cisions can lead to much larger performance improvements than could be achieved by improving the optimizer itself.

The benchmarks in Figure 3 are sorted in order of increasing running time, confirming that our technique has a more dramatic effect on shorter running programs (near the top), as would be expected because our technique makes profiling information available earlier in the program's lifetime. However, our approach improves performance for the longer-running programs as well; the running time of the bottom 6 benchmarks is significant, ranging from $16 - 29$ seconds on a 3.0 GHz processor, and our technique produces speedups ranging from $5.7\% - 15.0\%$.

Figure 4 shows the `warmup` behavior of our system over time for the single input scenario, averaged over all benchmarks. Each point along the x-axis represents a program run. The y-axis shows percentage improvement relative to the base system. The circle shows the *average* performance over all benchmarks, and the error bars show one standard deviation.

During the first execution, the repository is empty, thus the behavior is essentially identical to the base system;[3] however, performance is slightly degraded due to the time spent writing to the profile repository.

As additional program executions occur, the compilation bound is increased (see Section 3.4) allowing more aggressive compilation decisions to be made. Average performance increases through roughly the first 15 execution, at which point it levels out at around 15%.

5.4 Dual input scenario

This section presents a more challenging scenario by alternating between the small and large program inputs. Figure 5 presents the `steady state` performance results for this scenario, using the same format as Figure 3. Despite training the repository using two separate program inputs, the performance gains for many of the benchmarks are nearly as large as when trained on a single input.

Overall, the performance results are similar to Figure 3, but with the longer running programs having slightly smaller gains. These results confirm the success of our algorithm for selecting online strategies (Section 3.3), demonstrating its ability to construct an online strategy that is effective in the presence of methods with bimodal running times. The performance gains of the large inputs are reduced because aggressive compilation needs to be delayed to ensure reasonable performance for the small inputs; however, the system still shows improvements in the range of 5% for the longest running programs.

The `compress` benchmark is somewhat of an anomaly, achieving large performance gains in both the single and dual input training scenarios, ranging from $41\% - 49\%$. These gains are caused primarily because `compress` spends the majority of its time in a small number of long-running methods. Once these methods are invoked, they execute for a long time without exiting, so making a wise initial compilation choice is critical. This effect would be reduced in a system that performs on-stack replacement (OSR) [10], but would not disappear entirely. A system performing OSR would still need to identify the long-running methods, and pay the penalty of performing the on-stack replacement. Our system would avoid these costs by making a better initial compilation choice.

Figure 6 shows the `warmup` behavior of our system over time for the dual input scenario. The format of the graph is the same as Figure 4; Each point along the x-axis represents a program run, the circle shows *average* performance over all benchmarks, and the error bars show one standard deviation.

In this scenario the program inputs alternate between the small and large inputs. Our technique produces larger im-

[3]Recall that when a precomputed strategy does not exist for a method, the system defaults to the original online behavior.

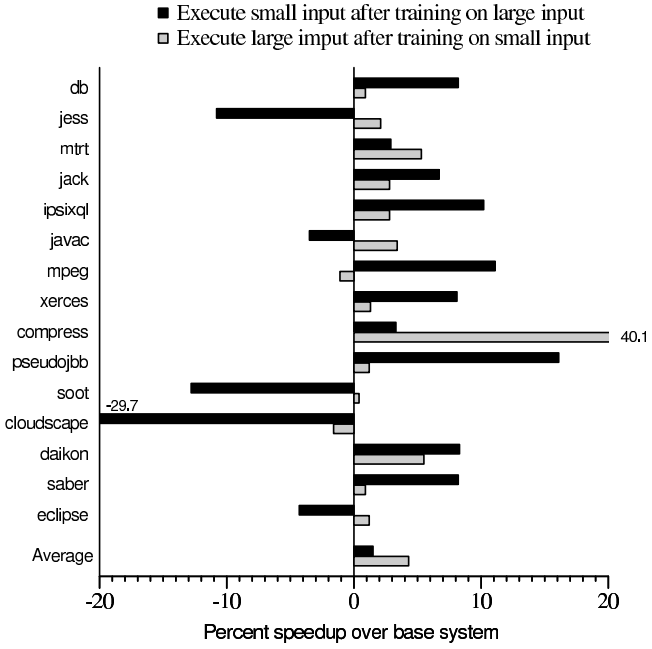

Figure 7: Phase shift scenario. Performance of first execution with new input.

provements for the small inputs, thus the average performance alternates between roughly 7% and 13%.

5.5 Phase shift scenario

Our technique improves performance by assuming that past behavior will repeat itself in the future. This section explores the performance of our system when this assumption does not hold because program behavior changes. For this scenario, each program was executed 10 times with one input, then 10 times with the other input. Both phase shifts were explored (small to large, and large to small).

The `steady state` behavior for this scenario is not interesting because it would converge on the same performance as the dual input scenario. Instead, Figure 7 presents performance of the first execution *after* the change in input size. The black bar shows the performance of the small input after training on the large input; the gray bar shows the performance of the large input after training exclusively on the small input.

First, observe that almost all of the gray bars are positive, with an average improvement of 4.3%, showing the performance of the large inputs is improved even when trained on the small inputs. This is not surprising because when training on small inputs, our system usually performs some compilations more aggressively than the base system, and this aggressiveness is beneficial when executing the large inputs as well.

The main source of potential performance degradation for our system is when executing a small input after training on the large input (shown by the black bars). The aggressive compilation performed early in the execution will likely degrade the performance of short running programs.

However, the performance of the small inputs actually *improved* for the majority of the benchmarks as well, despite being trained on the large input. This is largely attributed

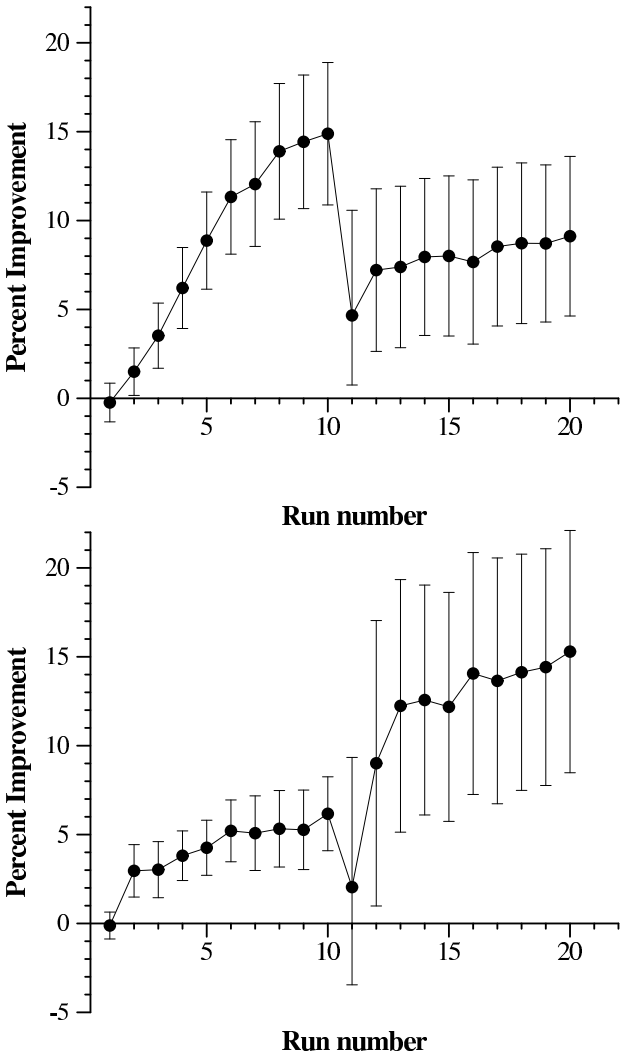

Figure 8: Phase shift scenario, behavior over time. Performance is averaged over all benchmarks. The top graph shows shifting from the small input to the large input. The bottom graph shows shifting from the large input to the small input.

to the compilation bound, which prevents the system from being too aggressive early in the program execution. Even when the profile repository suggests that a method is strictly long-running, the compilation bound forces a delay of the highest levels of optimization, thus methods start out at lower optimization levels. This gradual approach creates potential benefit even when the program exits earlier than expected.

Five benchmarks showed slowdowns when switching to the small input, with cloudscape being the largest by far at -29.7%. Although these slowdowns are not ideal, they are not unexpected in the presence of a phase shift, similar to a cache miss or a page fault. Like most profile-guided optimizations, our system strives to improve *average* performance by potentially reducing the performance of unlikely scenarios. However, bounding compilation keeps even the worst-case performance manageable. Moreover, this performance is truly unlikely, as it is expected only in the *first*

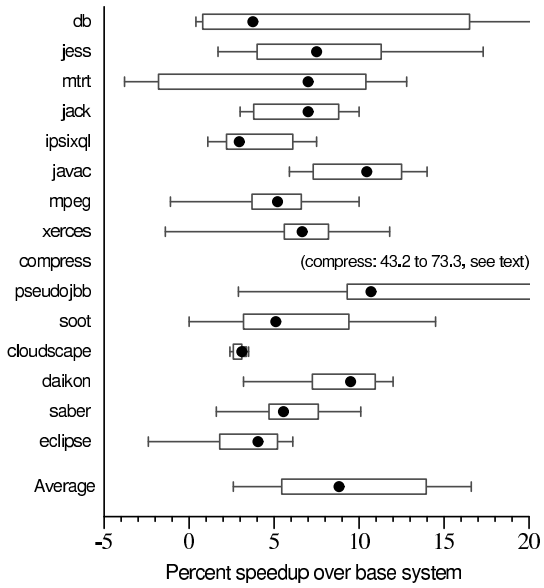

Figure 9: Multiple input scenario, steady state performance.

execution with a new input. After executing the new input once, this repository will be trained on this input and react accordingly, as shown in the next figure.

Figure 8 shows the average performance over time in both of the phase shift scenarios. The first graphs shows the shift from small to the large input, while the second graph shows the reverse. Each point shows the performance average over all benchmarks, and the error bars show one standard deviation.

The dip in the curve shows when the input changed. As discussed previously, the *average* performance over all benchmarks is always positive, even for the first run of the new input. The second run with the new input is already substantially improved because the repository has been trained on the new input, and performance continues to improve as execution of the new input continues.

5.6 Multiple input scenario

For the final usage scenario, we expanded the set of inputs to create a more challenging execution scenario. Each benchmark was augmented with 8 additional inputs, yielding 10 inputs total. Table 2 presents the running time of the programs with these inputs, sorted from shortest to longest, when run using our base version of J9. The inputs were chosen somewhat arbitrarily, but with some attempt to produce a reasonable distribution of running times that ranged beyond the original *small* and *large* inputs.

For this execution scenario, a random input ordering was selected for each benchmark, and this ordering was repeated for 50 total executions of the benchmark. Figure 9 shows the `steady state` performance for this scenario. Each benchmark has 10 performance results (one for each input). These 10 performance results are shown for each benchmark using a *boxplot*. The black circle represents the median performance of the 10 inputs. The rectangle represents the middle two quartiles (in our case, data points 3–8), and the hash marks at the end of the horizontal line represent the minimum and maximum performance. As with the previous steady-state graphs, the x-axis represents speedup over the base system.

The overall performance trend is overwhelmingly positive, even when a wide range of inputs are used. Our system outperforms the base system for all but 7 of the 150 program/input pairs trained and evaluated using this methodology (15 programs with 10 inputs each), with the largest degradation being 3.8%. The median improvement ranged from 3.1 to 43.3%, with a average of 8.2% across the benchmarks.

The performance improvement for `compress` was large for all inputs (as discussed in 5.4), ranging from 43% to 73%; its row in Figure 9 is empty because the entire boxplot falls beyond the range of the chart.

Figure 10 shows average performance of the system over time while executing in this multi-input scenario. Each program was executed 50 times, using the randomly preselected input order. Each point shows the performance average over all benchmarks, and the error bars show one standard deviation.

5.7 Repository characteristics

Tables 3 and 4 present characteristics of the profile repository for the benchmarks used in this paper.

Table 3 reports size characteristics of the repository for each benchmark after being warmed up. Data is reported for the small, large, and dual input scenarios. The repository is generally smaller for the small inputs because fewer methods are identified as hot and recorded in the repository. The table shows the number of methods for which profile data is recorded, both as an absolute number and as a percentage of all methods executed during the program run. The second column for each input scenario shows the total size of the repository, in kilobytes and as a percentage of the program size (size of the classfiles that are executed at runtime, as reported in Table 1).[4]

Table 4 shows a breakdown of the overhead incurred when reading and updating the repository. For each input scenario, the table reports three quantities, all reported as a percentage of the total execution time.

1. **Startup I/O:** time spent reading the precomputed online strategies from the repository at program startup.
2. **Shutdown I/O:** time spent reading and writing the raw profile data repository, as well as writing the updated precomputed online strategies.
3. **Analyze:** time spent performing profile analysis (the algorithm presented in Section 3.3).

The amount of time spent updating the repository was small, generally between 0.1%–0.3% of total execution time. The largest overheads are incurred for the short running inputs when trained in the dual training scenario; the average overheads were on the order of 0.5% for each component, with the largest individual overhead being 1.9% for profile analysis of `cloudscape-small`. Larger overhead is incurred in this scenario because the longer running program writes more data to the repository; the short running program needs to process the larger quantity of data, thus the

[4]For the dual input columns, percentages are computed relative to the large input.

Prog.	Input Number									
	1	2	3	4	5	6	7	8	9	10
db	0.3	0.4	0.6	0.9	1.6	3.2	10	19	23	28
jess	0.3	0.5	0.6	0.9	1.3	2.0	2.7	4.4	6.0	7.7
mtrt	0.3	0.5	0.6	1.1	1.7	2.2	2.9	3.6	4.8	6.0
javac	0.5	0.5	0.6	0.8	0.8	0.9	1.8	3.2	3.3	5.3
mpegaudio	0.1	0.1	0.6	1.3	2.1	2.5	3.0	3.5	3.7	3.8
compress	0.6	1.1	2.3	2.8	2.9	3.6	4.0	5.0	5.1	6.1
ipsixql	0.8	1.6	1.8	2.3	3.7	4.0	5.0	7.5	10	16
xerces	0.8	0.9	1.0	1.1	1.3	1.5	1.6	1.7	2.1	5.7
pseudojbb	0.8	0.8	1.6	2.2	4.4	5.3	5.4	8.4	10	13
soot	1.5	1.9	2.0	5.2	7.0	12	17	18	29	45
daikon	1.5	2.7	5.0	6.7	8.2	8.8	9.4	11	12	14
saber	2.6	2.7	4.2	5.0	6.0	6.4	8.0	9.6	11	16
cloudscape	1.8	8.3	11	13	14	16	18	20	22	23
eclipse	5.9	8.8	8.9	15.5	18.0	18	19	19	20	22

Table 2: Running times in seconds for programs with the expanded set of 10 inputs.

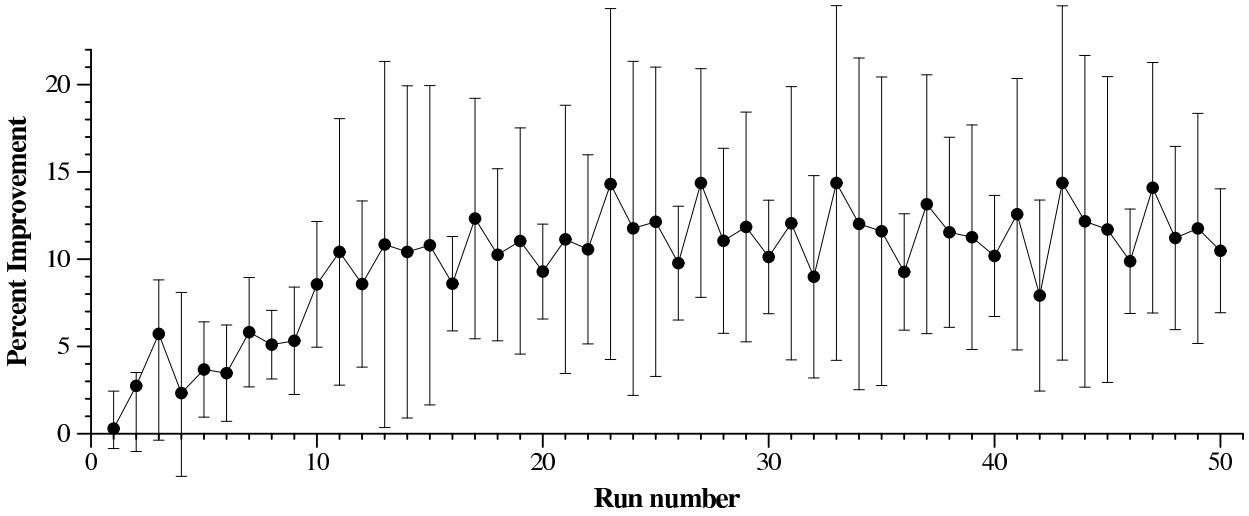

Figure 10: Multi input scenario, behavior over time. Performance averaged over all benchmarks.

Program	Small Input		Large Input		Dual Input	
	Methods	Size (KB)	Methods	Size (KB)	Methods	Size (KB)
jess	191 (14%)	19 (1%)	235 (18%)	28 (2%)	271 (20%)	33 (3%)
mtrt	262 (24%)	30 (3%)	237 (22%)	29 (3%)	266 (24%)	32 (3%)
jack	211 (19%)	21 (2%)	288 (26%)	33 (3%)	308 (28%)	35 (3%)
javac	443 (27%)	58 (5%)	732 (44%)	112 (9%)	772 (46%)	117 (10%)
mpegaudio	164 (15%)	16 (1%)	195 (18%)	21 (2%)	233 (22%)	26 (2%)
compress	81 (9%)	7 (0%)	70 (7%)	6 (0%)	107 (12%)	10 (1%)
ipsixql	119 (12%)	10 (1%)	136 (13%)	16 (2%)	173 (17%)	20 (2%)
xerces	233 (15%)	26 (2%)	327 (21%)	40 (3%)	360 (23%)	44 (3%)
pseudojbb	416 (28%)	50 (5%)	408 (28%)	54 (5%)	450 (31%)	60 (6%)
soot	381 (17%)	38 (2%)	1379 (49%)	174 (10%)	1425 (51%)	182 (10%)
saber	571 (16%)	65 (2%)	1093 (23%)	139 (4%)	1183 (25%)	152 (4%)
daikon	737 (26%)	81 (4%)	856 (57%)	101 (5%)	945 (63%)	114 (6%)
cloudscape	326 (5%)	35 (0%)	3079 (33%)	350 (5%)	3073 (33%)	358 (6%)
eclipse	675 (6%)	78 (1%)	2413 (12%)	312 (2%)	2597 (13%)	341 (2%)

Table 3: Repository Size Characteristics

	Single Input			Dual Input		
Program	Start I/O	Exit I/O	Anl	Start I/O	Exit I/O	Anl
db-small	0.1	0.2	0.1	0.3	0.3	0.2
jess-small	0.1	0.3	0.2	0.2	0.6	0.4
mtrt-small	0.2	0.4	0.2	0.2	0.5	0.3
jack-small	0.1	0.3	0.1	0.2	0.5	0.4
javac-small	0.2	0.4	0.2	0.5	1.2	0.9
mpegaudio-small	0.1	0.3	0.1	0.1	0.5	0.3
compress-small	0.0	0.1	0.1	0.1	0.2	0.1
ipsixql-small	0.1	0.2	0.1	0.1	0.4	0.3
xerces-small	0.1	0.3	0.2	0.2	0.4	0.3
pseudojbb-small	0.0	0.1	0.1	0.0	0.1	0.1
soot-small	0.2	0.3	0.2	0.5	1.5	1.4
saber-small	0.1	0.2	0.1	0.2	0.6	0.4
daikon-small	0.1	0.3	0.2	0.2	0.5	0.4
cloudscape-small	0.1	0.2	0.1	0.8	1.9	1.0
eclipse-small	0.1	0.1	0.1	0.2	0.5	0.2
Mean small	0.1	0.2	0.1	0.3	0.6	0.4
db-large	0.0	0.0	0.0	0.0	0.0	0.0
jess-large	0.0	0.1	0.1	0.1	0.2	0.1
mtrt-large	0.1	0.2	0.1	0.1	0.2	0.1
jack-large	0.1	0.1	0.1	0.0	0.1	0.1
javac-large	0.1	0.2	0.2	0.1	0.3	0.2
mpegaudio-large	0.0	0.1	0.1	0.0	0.1	0.1
compress-large	0.0	0.0	0.0	0.0	0.0	0.0
ipsixql-large	0.0	0.1	0.1	0.0	0.1	0.1
xerces-large	0.1	0.2	0.1	0.1	0.2	0.2
pseudojbb-large	0.0	0.0	0.0	0.0	0.0	0.1
soot-large	0.0	0.1	0.1	0.0	0.1	0.1
saber-large	0.0	0.1	0.1	0.0	0.1	0.1
daikon-large	0.0	0.1	0.1	0.1	0.2	0.1
cloudscape-large	0.1	0.2	0.1	0.1	0.2	0.1
eclipse-large	0.1	0.2	0.1	0.1	0.2	0.1
Mean large	0.0	0.1	0.1	0.0	0.1	0.1
Mean all	0.1	0.2	0.1	0.1	0.4	0.3

Table 4: Overhead of updating repository. Columns show percent of run time spent 1) reading the precomputed online strategies at VM startup, 2) performing repository I/O at VM shutdown, and 3) performing the profile analysis at VM shutdown.

overhead increases. However, as shown in Section 5.4, our technique provides the largest benefit for the short running programs, so the repository overhead is more than compensated for. All performance data reported included the overhead of reading and updating the repository.

The overheads in this section would likely be reduced with a more highly tuned implementation of our system. The repository is stored in plain text format and no effort was made to compress the data or speed up I/O. In addition, overhead can by reduced or limited by using the techniques discussed in Section 2.4, such updating only a subset of the data on each program execution.

6. RELATED WORK

We are not aware of any virtual machine that automatically persists profile data across multiple runs of an application, and uses this information together with online profiling information at runtime to improve performance.

Systems that perform optimization based on offline profile data have existed for many years [18]. However, requiring a manual training step drastically reduces the chance that such a technique will be used by a typical developer. In addition, all of these systems assume a clear distinction between training and production runs, and none addresses how a virtual machine would combine offline information together with online information being collected at runtime.

Sandya [16] describes a preliminary study combining offline and online profile data to guide selective optimization in the Hotspot JVM. Their work is similar to ours in that it combines both online and offline profile data to make online optimization decisions in a VM. However, there are a number of differences. Their work uses explicit training runs to collect the offline data, requires manually specifying a "confidence" level in the offline data, and does not discuss training on multiple inputs. The focus of our work is on avoiding the manual training step, and ensuring good performance in the presence of multiple inputs. Finally, their work combines offline and online data using threshold-based heuristics, with only preliminary data reported for the SPECjvm98 benchmarks. Cross training is performed from size 10 to size 100 (train on size 10 and run with size 100) but not the reverse. In our work, we were unable to find a simple, threshold-based policy that would provide acceptable performance for a wide range of benchmarks, input sizes, and execution scenarios.

Krintz and Calder [13] describe annotating Java bytecode to identify hot *priority methods* that should be optimized immediately at higher optimization levels. In addition to requiring a training step, this work does not generalize to programs that have a wide range of inputs. Their technique specifies that methods are always optimized at a fixed optimization level, without considering online profiling information; if a program has two inputs, one short running and one long running (as evaluated in Section 5.4) their fixed strategy could perform poorly, either over-compiling for the short running programs, or under-compiling in the long-running ones.

Childers et al. [4] describe an architecture for a continuous compilation, *CoCo*, which includes a dynamic optimizer and the ability to perform optimization across multiple runs. Their architecture assumes a static compilation model, and makes use of profile data collected via explicit training runs. The implementation of their work focuses mostly on modeling the impact of compiler optimization.

Cascaval et al. [22] presents a framework for Continuous Program Optimization (CPO) across multiple levels of the software stack (application, VM, OS, hardware), and performs optimization over multiple executions of the program. The initial instantiations of the archicture have focused on offline profiling, collecting events at the operating system and hardware level. Our work is online in a JVM, and can be considered a "Online VM agent" in their architecture.

Other systems have performed *ahead-of-time compilation*, or *static compilation* of Java [17], where compilation is performed prior to program execution, thus avoiding the overhead of performing compilation at runtime. This approach has a number of advantages, such as avoiding the need to perform compilation at runtime; however it has yet to become popular for the Java programming language for a number of possible reasons. First, it changes the execution model, introducing security concerns by eliminating the process of bytecode verification; modifying the compiled code on disk would circumvent all of Java's safety guarantees. Our technique does not require the user to accept a new execution model, and does not introduce any security concerns. Second, static compilation involves a number of technical

challenges for dynamic language with features such as dynamic class loading and reflection; the code generated by static compilers for Java is often substantially lower quality than that generated by the dynamic JIT compiler. In addition, selective optimization is only example instantiation of using the profile repository to improve performance. Even systems that perform static compilation can benefit from the use of online and offline profile data, thus can benefit from many of the ideas presented in this work.

7. DISCUSSION

Any time new functionality is added to a virtual machine, the designers must decide whether the benefits warrant the cost of the additional complexity to the system. Compilers and virtual machines are already incredibly complex and their behavior is difficult to understand and debug. Having the VM persist profile data across runs adds yet another level of complexity, but after spending time working with our system, we were pleasantly surprised in this regard.

The repository turned out to be an effective debugging aid. At any given time, the profile data provide a summary of what happened in previous runs, and the precomputed online strategies describe what is about to happen in future runs. Several times when searching for a bug, we ended up not even needing to execute the program because looking in the repository was sufficient.

8. CONCLUSIONS

This paper presented an architecture for a persistent profile repository, allowing a virtual machine to remember profile data across multiple runs of an application. This approach widens the scope of profile-guided optimizations, allowing them to be performed over the lifetime of an application, rather than being restricted to a single execution.

We described in detail how this architecture can be used to improve selective optimization. Our results demonstrate substantial performance improvements, with the average performance ranging from 8.8% – 16.6% depending on the execution scenario.

In future work we plan to use the profile repository to improve additional optimizations, such as feedback-directed inlining. Using a profile repository to augment these optimizations should increase their benefit by allowing them to be performed earlier in the applications lifetime.

9. ACKNOWLEDGMENTS

We would like to thank Vivek Sarkar, Michael Hind, and Evelyn Duesterwald for their support of this work. We also thank Michael Hind for providing feedback on an earlier draft of this paper, and the anonymous reviewers for their helpful comments.

This work was supported in parts by the Defense Advanced Research Project Agency under contract NBCH30390004.

10. REFERENCES

[1] 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)*.

[2] 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.

[3] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 37(11):111–129, Nov. 2002. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[4] B. R. Childers, J. W. Davidson, and M. L. Soffa. Continuous Compilation: A new approach to aggressive and adaptive code transformation. In *NSF Workshop on Next Generation Software*, 2003.

[5] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices*, 35(5):13–26, May 2000. In *Conference on Programming Language Design and Implementation (PLDI)*.

[6] IBM Cloudscape relational database management system. http://www.cloudscape.com.

[7] Colorado Bench. http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.

[8] The Daikon dynamic invariant detector. http://pag.csail.mit.edu/daikon.

[9] Eclipse Foundation. Eclipse. http://www.eclipse.org.

[10] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization (CGO)*, pages 241–252, 2003.

[11] 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.

[12] U. Hölzle and D. Ungar. A third generation SELF implementation: Reconciling responsiveness with performance. *ACM SIGPLAN Notices*, 29(10):229–243, Oct. 1994.

[13] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 156–167, New York, NY, USA, 2001. ACM Press.

[14] 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.

[15] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. Johnson, A. Kershenbaum, and L. Koved. Saber: Smart analysis based error reduction. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 252–262, July 2004.

[16] S. M. Sandya. Jazzing up jvms with off-line profile data: does it pay? *SIGPLAN Not.*, 39(8):72–80, 2004.

[17] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. *ACM SIGPLAN Notices*, 35(10):66–82, Oct. 2000. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[18] M. D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). *ACM SIGPLAN Notices*, 35(7):1–11, July 2000.

[19] Soot, a Java Bytecode Analysis and Transformation Framework. http://www.sable.mcgill.ca/software/#soot.

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

[21] 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)*.

[22] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and environment monitoring for whole-system characterization and optimization. In *PAC2 Conference on Power/Performance interaction with Architecture, Circuits, and Compilers*, 2004.

[23] Xerces2 Java Parser Readme. http://xml.apache.org/xerces2-j/index.html.