Technical Report DCS-TR-469 , Department of Computer Science, Rutgers University

Online Instrumentation and Feedback-Directed Optimization of Java

Matthew Arnold
marnold@cs.rutgers.edu
Rutgers University, Piscataway, NJ, 08854
*IBM T.J. Watson Research Center, Hawthorne, NY, 10532

Abstract

The overhead involved in collecting fine-grained profiling information makes feedback-directed optimizations difficult to perform online at runtime. As a result, the vast majority of work in offline feedback-directed optimization is not yet being applied in online systems. This paper describes the design and implementation of a fully automatic online approach for performing instrumentation and feedback-directed optimization. Our approach uses instrumentation sampling to reduce the overhead of instrumentation, thus eliminating many of the limitations present in existing online systems. Several online feedback-directed optimizations are described, including a novel algorithm for performing feedback-directed splitting. Our experimental results show improvements in peak performance of up to 20% while overhead remains low, with no individual execution being degraded more than 2%.

1 Introduction

The dynamic nature of the Java$^{\mathrm{TM}}$ programming language makes it a natural candidate for dynamic and feedback-directed optimizations. Many of today's Java Virtual Machines (JVMs) use coarse-grained profiling information to focus optimization efforts on program hot spots [3,13,26,28]. Although this approach can result in large speedups for short running applications, it does little to improve the performance of long running programs that easily amortize the cost of optimizing all methods.

For long running applications, such as server applications, the most substantial performance improvements will come from *feedback-directed* optimizations (also known as *profile-guided* optimizations), where fine-grained profiling information is collected and used to fine-tune the decisions made by the optimizing compiler. Several systems [16,22,23] have shown that performance can be improved substantially by exploiting invariant runtime values; however, these systems were not fully automatic and relied on programmer directives to identify regions of code to be optimized. Other fully automated work in feedback-directed optimization has used *offline* profiles collected during a separate training run (ex., [12, 19, 24, 27]). Although the resulting speedups are often promising, this approach fails in scenarios where a) it is impractical to collect a profile prior to execution, or b) the application does not behave like the training run.

*Matthew Arnold was an intern at Watson for much of the duration of this work.

Performing profiling and optimization *online*, during the same run, is an attractive approach because it avoids the previously mentioned drawbacks of offline profiling. Unfortunately, using online profiles to guide optimization has limitations of its own because the amount of work that must be performed at runtime is increased, creating the potential for *degrading* performance rather than improving it.

The overhead of online instrumentation is a real concern and is one of the reasons today's JVM's perform only limited forms of feedback-directed optimizations [3, 13, 26, 28]. Optimizations that are currently being used online are usually based on profiles that can be collected easily with low overhead. Some online systems, such as Dynamo [7], are designed to identify when performance is being degraded so that profile-guided optimizations can be disabled for the remainder of execution.

Instrumentation sampling [5] is a compiler transformation that reduces the runtime overhead of executing instrumented code. This paper describes an approach for using instrumentation sampling online to drive feedback-directed optimizations in an adaptive Java Virtual Machine. We show that with minimal overhead, instrumentation sampling can be used effectively to reduce the overhead of collecting online instrumented profiles. By allowing a wide range of traditionally offline instrumentation techniques to be used at runtime, one the biggest obstacles to performing feedback-directed optimizations online is eliminated. We demonstrate, using several examples of feedback-directed optimizations, that our online approach can be used to effectively improve the performance of long running Java applications, without sacrificing the performance of short running applications.

The contributions of this paper include

- **Online Strategy** We describe the design and implementation of a new approach for performing online instrumentation and feedback-directed optimization. Our online instrumentation strategy uses *instrumentation sampling* [5], allowing a wide variety of previously offline instrumentation techniques to be used online.
- **Instrumentation** We describe the inclusion of one example instrumentation technique, intraprocedural edge counters, into our system.
- **Feedback-Directed Optimizations** We describe the inclusion of several online feedback-directed optimizations, including
  1. a novel algorithm for performing feedback-directed splitting

2. several other optimizations that benefit from runtime information, such as basic block reordering, inlining, and loop unrolling.

- **Evaluation** We validate the techniques presented with a detailed experimental evaluation of their implementation in the Jikes Research Virtual Machine[1] We show that our techniques reliably improve performance of long running applications without degrading the performance of short running applications. We show a maximum improvement of 20.9% with an average improvement of 8.3% for long running versions of the SPECjvm98 benchmark suite. We also show up to 8.9% throughput improvement for the SPECjbb2000 server benchmark.

2 Background

Many virtual machines have mechanisms in place for focusing optimization efforts on program hot spots [3, 13, 18, 26, 28]. In most cases, coarse grained profiling is used to determine methods that are frequently executed, and an optimizer is invoked on these methods.

The goal of feedback-directed optimization (FDO) is to improve performance further by using profiling to direct not only *what* methods to optimize, but also *how* to optimize them. Fine-grained profiling information can be used to guide the decisions made by the optimizing compiler to exploit common execution patterns, ultimately improving the quality of the generated code.

2.1 Difficulties of Online FDO

Performing feedback-directed optimization online is more difficult than performing them offline for two main reasons.

1. **Overhead** There are several sources of overhead involved in performing feedback-directed optimizations online. For example, there is overhead for 1) collecting the profiling information, 2) examining the profile data and making decisions based on it, and 3) performing the actual profile-guided optimizations. Care must be taken during all of these steps or performance may be reduced rather than improved.

2. **Profile accuracy and availability** When using offline profile information, the profile is usually assumed to be free, accurate, and available prior to execution. With online profiling, none of these assumptions are true. Instead, the accuracy and availability of profile data are dependent on 1) what is instrumented, 2) when instrumentation is performed, and 3) how long instrumentation is performed.

   There is no simple solution for these problems that is best in all scenarios; instead answering them involves balancing a number tradeoffs. For example, instrumenting for a shorter duration will reduce instrumentation overhead and allow the profile to be available sooner, however, it may result in a less accurate profile. Similarly, instrumenting soon after the program begins execution will allow the profiled information to be available sooner, but may come at the cost of profile accuracy if the program's startup behavior is not representative of its long running behavior.

[1] Jikes RVM is an open source version of the Jalapeño Research Virtual Machine [1, 3].

2.2 Existing Online Strategies

Systems that use online profiling information to drive feedback-directed optimizations generally collect their profiles in one of 3 ways:

1. **Instrument early during unoptimized execution**

   Hotspot [26] and the IBM DK 1.3.0 [28] collect fine-grained information about each method during interpretation, before the method has been optimized. The advantage of this approach is that the overhead of instrumentation is likely to be low given the already poor performance of the unoptimized (or interpreted) code. Another advantage is that the profile will be available when the method is first recompiled, allowing feedback-directed optimization to occur as early as possible.

   However, there are several limitations to profiling unoptimized code. First, it allows methods to be profiled only during their early stages of execution, since profiling ends once optimization occurs. This approach can be ineffective, or even counter-productive, for methods whose dominant behavior is not exercised during the early execution of the method.

   To identify rare code for driving partial method compilation [31] Whaley proposed a 3 stage model where instrumentation is avoided during interpretation, and is inserted in stage 2 when the method is compiled with lightweight optimizations. Although this approach may help in some cases, it is not a general solution. Initialization time may vary for different applications and phase shifts may occur; this approach provides no way of collecting a profile once a certain amount of time has passed.

   A second difficulty of instrumenting unoptimized code is that it makes certain profiling information more difficult to collect. For example, optimizing compilers often perform aggressive method inlining, which can drastically change the structure of a method. Determining certain information, such as the hot paths through the inlined code, can be non-trivial when using a profile obtained prior to inlining, using the unoptimized code.

   The MRL VM [13] also uses coarse grained instrumentation to trigger recompilation, but to our knowledge does not use fine grained instrumentation to perform FDO.

2. **Instrument optimized code**

   Profiling optimized code addresses the shortcomings described above, but suffers from a new problem, that the overhead of instrumentation can be substantial. Because of this overhead, few systems currently use this approach.

   The IBM DK 1.3.0 [28] is the most robust publicly described JVM that performs online instrumentation of optimized code. To keep overhead low, their system uses *code patching* to dynamically remove instrumentation after it has executed for a fixed number of times.

   The main disadvantage of code patching is that it places limitations on the types of instrumentation that can be used; specifically, instrumentation can be inserted only if it can be effectively removed by patching the executable. The overhead and scalability of this technique remain unclear, particularly for scenarios that require substantial code patching, as would be the case when applying more complex instrumentation techniques, or when applying several different types of instrumentation together at once. In these scenarios, execution may remain partially degraded even after patching occurs.

A second limitation of this approach is that to keep overhead low, profiles must be collected in short bursts of time. For profiling information that is 1) not strongly peaked, or 2) varies over time, this technique could easily produce an inaccurate profile. Finally, although less significant than the above, code patching is architecture specific and may introduce complexities such as maintaining cache consistency.

In some cases instrumentation is simply applied directly. Kistler [21] described using online instrumentation in an Oberon system. Their system directly applied instrumentation and simply computed how long the program needed to run to recover the time lost due to instrumentation overhead. For applications that do not run long enough, performance could be severely degraded. Exception Directed Inlining (EDO) [25] instruments exception handling to profile exception paths from the throw to the catch. However, this instrumentation is likely to be lightweight given the infrequent occurrence and relative expense of handling the exception.

3. **Sample throughout execution**

   Certain kinds of profiling information can be collected by means other than instrumentation. For example, the Jikes RVM's adaptive system [3] profiles the call graph by periodically sampling the call stack at regular time intervals. This approach is attractive because it has low overhead and can run throughout the application's lifetime. The main limitation is that there are many types of profiling information for which there is no obvious sampling solution. The IBM DK 1.3.0 [28] also uses a timer-based sampling mechanism to select methods for optimization, but to our knowledge does not use it to drive feedback-directed optimizations.

3 Design

This section describes our design for performing online instrumentation and feedback-directed optimization in a VM. Because of the variety of techniques for incorporating profiling into an online system, and the associated tradeoffs, it is important to start by making clear the goals of the online system being designed.

3.1 Goals

Our online approach assumes an execution environment where a lighter weight mechanism is used to identify hot sections of code and promote them to higher levels of optimization. Our purpose for collecting fine-grained profiling information is to collect additional information that can be used to improve the decisions made by the optimizing compiler, further improving performance Our goals for performing feedback-directed optimizations are as follows, listed in order of importance beginning with the highest priority first.

1. **Improve the performance of long running applications**.

   Long running applications, such as server applications, are prime candidates for feedback-directed optimization. Although virtual machine technology has begun taking advantage of feedback-directed optimizations, most rely on simple profiling techniques and often collect profiles only during the first few moments of execution.

   Our primary goal is to improve the peak performance of long running applications, and to do so reliably, regardless of factors such as how the application behaved during startup, and whether phase shifts have occured since the application was launched.

   Achieving peak performance requires a profiling mechanism that is flexible enough to support a wide range of feedback-directed optimizations, including those traditionally used offline. A second requirement is the ability to profile for more than just the first few moments of execution. It is unacceptable for an application to perform poorly for a long period of time (possibly several days) because the behavior observed during program startup was not representative of the entire execution.

2. **Maintain startup performance.**

   Peak performance should be obtained without compromising the system's usefulness as a general VM that is able to provide high performance for both short and long running applications.

3. **Improve performance as early as possible.**

   With all other factors constant, reaching high performance earlier than later is preferable. However, achieving high performance sooner is not worth compromising the level of performance that is eventually reached, or substantially degrading startup performance.

3.2 Instrumentation Sampling

Achieving peak performance, our primary goal, involves putting to use the entire body of traditionally offline feedback-directed optimizations. To remove many of the existing limitations of instrumenting code online, our design uses the *Full-Duplication instrumentation sampling framework* [5], a fully automatic compiler transformation that transparently reduces the overhead of executing instrumented code.

The key idea of the Full-Duplication technique is as follows: when a method is instrumented the body of the method is duplicated and all instrumentation is inserted into the duplicated code. The original version of the method is only minimally instrumented at sample points that allow control to be transfered into the duplicated code. Samples are taken on regular intervals and control is transfered into the duplicated (and instrumented) code for a finite amount of time. A counter-based sampling mechanism is used to provide a flexible sample rate and deterministic sampling.

This technique has been shown to have low overhead and high accuracy for several examples of instrumentation. It is also flexible, allowing wide range of instrumentation techniques to be used without modification, making it easy to apply existing instrumentation techniques online.

The main disadvantage of the Full-Duplication technique is that the entire method is duplicated, increasing both space (by a factor of 2) and compile time (by roughly %30 [5]). This compile time increase affects bottom line performance when compilation occurs at runtime, therefore candidates for online instrumentation must be chosen carefully.

3.3 Online Strategy

Although instrumentation sampling reduces the overhead of executing instrumented code, a number of decisions must still be made when performing instrumentation online. This section describes our decision making strategy for performing online instrumentation and profile-guided optimizations. The job of the decision making strategy is to determine

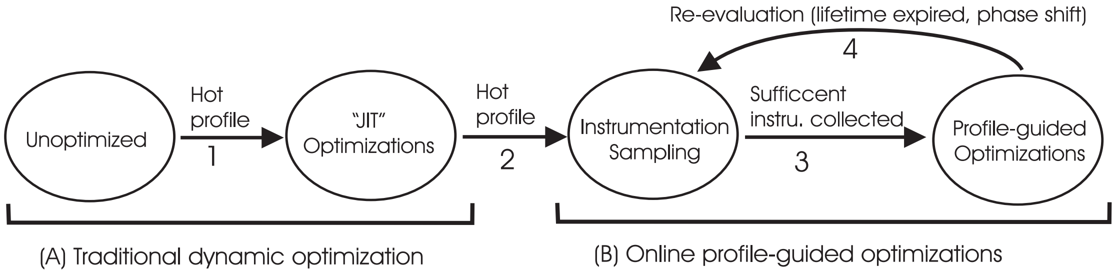

Figure 1: A graphical representation our online strategy for performing profile-guided optimization. Each circle represents the possible state of a method. The transitions (labeled 1–4) are described in detail in Section 3.3.

when, and on what methods, instrumentation and feedback-directed optimization should be performed to minimize overhead while maximizing performance gains, ultimately trying to achieve the goals described in Section 3.1.

A high level view of our online strategy is shown Figure 1. Each circle represents a possible optimization state of a method. Each arrow (labeled 1–4) represents a transition (via recompilation) from one state to the next. Each transition is described below:

1. Execution begins with no instrumentation and the underlying adaptive system recompiles methods as it sees fit, possibly using multiple optimization levels. Avoiding instrumentation during the early executions of a method reduces the risk of degrading performance for short running applications.
2. After a method has been optimized "statically" (without feedback-directed optimizations) it continues to be monitored; if it remains sufficiently hot, it is instrumented using instrumentation sampling. By selectively applying instrumentation, heavily executed regions of code can be profiled while recompilation is kept to a minimum. Because the instrumented methods are known to be heavily executed, instrumentation sampling is needed to maintain competitive performance, allowing execution to continue at near fully optimized speed [5] during the instrumentation period.
3. After the instrumented method has run long enough to collect sufficient profiling information, the method is recompiled again and feedback-directed optimizations are applied. Exactly how long the instrumented code should be run depends on several factors, including the type of instrumentation being performed and the sample frequency of the instrumentation sampling. Speed of convergence is our lowest priority design goal (see Section 3.1) so longer instrumentation periods are preferable to ensure that accurate profiles are collected.
4. As time passes, program behavior may change. To ensure that performance does not degrade slowly over time as the profile becomes out of date, profiles can be periodically reevaluated. Reevaluation can be triggered by simple techniques, such as a timer mechanism, or by more advanced phase-shift detection algorithms.

4 Implementation

The instrumentation infrastructure described in this paper is designed to be as platform- and VM-independent as possible. However, to validate the effectiveness of our proposed techniques they have been fully implemented in the Jikes Research Virtual Machine. We begin with a brief review of the Jikes RVM and its adaptive optimization system, followed by the modifications necessary to support our online instrumentation design.

4.1 The Jikes RVM

The Jikes RVM [1] is a Research Virtual Machine developed at IBM T.J. Watson Research Center. Jikes RVM is written almost entirely in Java, and takes a compile only approach (no interpreter). Methods are compiled with a non-optimizing *baseline* compiler upon their first execution, and an aggressive optimizing compiler is applied selectively by the adaptive optimization system [3].

The Jikes RVM's Adaptive Optimization System [3] is responsible for all aspects of online, adaptive optimization. The *Runtime Measurements* subsystem performs profiling throughout execution using a low overhead timer-based sampling mechanism to identify frequently executed methods. Periodically, sampled methods are passed to the decision making component, called the *controller*.

The controller uses a cost/benefit model to determine what action should be taken for each recompilation candidates it considers; as many decisions as possible are made using the model; ad-hoc parameters are avoided whenever possible. When considering whether to optimize a particular method, the viable choices are to 1) do nothing, or 2) recompile at one of Jikes RVM's 3 optimization-levels (O0, O1, O2). The controller estimates the cost and benefit of each potential recompilation choice, picking the choice that would minimize total execution time if the application were to run for twice as long as it currently has. The goal of the controller is to provides good performance for both short and long running applications.

To further improve performance of longer running applications, the adaptive system also performs *adaptive inlining*. The same sampling mechanism described above is also used to detect hot call edges. These hot edges are recorded so that they can be inlined if the callee method is recompiled in the future. Additionally, a hot call edge in a method that is already optimized at the highest level (O2) can trigger a recompilation of that method (from O2 $\rightarrow$ O2) for the sole purpose of incorporating the new inlining decision.

4.2 Modifying Jikes RVM's adaptive system

Incorporating the online approach described in 3.3 required only minor changes to the Jikes RVM adaptive system, the most important of which was introducing the notion of instrumentation and feedback-directed optimization to the cost/benefit model. The following is a detailed description of the necessary modifications.

1. The runtime measurements subsystem was modified to continue monitoring methods after they have been com-

piled at the highest non-FDO optimization level (O2). Without FDO there was no need to monitor such methods; however, with the addition of FDO, O2 methods need to be monitored so they can be considered for instrumentation by the controller.

2. The cost/benefit model of the controller was modified to incorporate the notion of instrumentation and feedback-directed optimization. When considering a method for optimization *without* FDO, the only options were to 1) do nothing, or 2) optimize at a higher optimization level. For FDO, a third choice is added to the model that consists of compiling with instrumentation inserted followed by recompiling with feedback-directed optimizations. Instrumentation is inserted at the highest optimization level (O2) so that the effects of optimization are reflected in the profile.

   The estimated cost of performing FDO consists of 3 parts: 1) the estimated cost of recompiling the method with instrumentation, 2) the estimated overhead of executing the instrumented code for the instrumentation duration, and 3) the estimated cost of compiling the method a second time to apply the FDO's. The expected benefit of the FDO choice is the estimated performance improvement that will occur after feedback-directed optimization has been performed.[2]

   If the benefit outweighs the cost, the controller notifies the recompilation subsystem that the method should be recompiled with instrumentation inserted.

3. After executing for some period of time, the profile collected by the instrumented method needs to be examined so that feedback-directed optimization can be applied. To make this possible, an "alarm clock" mechanism was added to the adaptive optimization system to allow the controller to reconsider instrumented methods after some amount of time has passed. The alarm clock enables the association of a particular event with a "time" in the future;[3] when the time arrives, the alarm clock sends the event to the controller for processing.

   Once the instrumented method is finished being compiled, the new code is installed and an alarm is set with an "examine instrumentation" event to notify the controller when it is time to examine the profile collected for this method. If not enough profiling information was collected (the given method never actually executed after it was instrumented) the alarm is reset and instrumentation continues executing for another time period.

4. When the controller decides that enough profiling information has been collected, the method is scheduled for feedback-directed optimization. Once compilation has completed, new calls to the method will transfer control to the highly optimized FDO code.

4.3 Modifying Jikes RVM's optimizing compiler

The Full-Duplication instrumentation-sampling technique is a general technique that can be implemented in a variety of ways, ranging from a source-to-source translation all the way to a binary-to-binary translation. This section describes some of the key design decisions for the implementation of the Full-Duplication transformation in the Jikes RVM optimizing compiler. For a full description, see [5].

An important decision not fully discussed in [5] is the relative placement of the Full-Duplication transformation in the optimization process. In our implementation, the Full-Duplication transformation occurs late in the optimization processes, as one of the last phases [1] of the machine-independent optimizations. Keeping the phase machine-independent was desirable for simplicity and reusability, however, performing the phase as late as possible was important for a number of reasons, including:

1. **Compile time overhead is reduced.** Because the Full-Duplication phase duplicates all of the code in the method, all remaining optimizations work on a method that is roughly doubled in size. By performing the Full-Duplication phase as late as possible, compile time is kept to a minimum because compilation occuring *after* the Full-Duplication phase is minimized.
2. **Interference with optimizations is reduced.** The Full-Duplication transformation increases the complexity of the control flow within the method, and therefore can indirectly introduce overhead if the new control flow interferes with any optimizations that follow. Performing the Full-Duplication transformation as late as possible minimizes the number of optimizations that may be affected by the transformation.
3. **Instrumentation phases are unaware of the Full-Duplication phase.** The Full-Duplication phase ensures that all instrumentation code is sampled by moving the instrumentation instructions out of the heavily executed original code, and into the infrequently executed duplicated code. The only requirement for instrumentation phases occuring *prior to* the Full-Duplication transformation is that they set an "instrumentation flag" on all instrumentation instructions they insert, making it possible for these instructions to be identified by the sampling framework. Keeping the instrumentation algorithms themselves decoupled from the sampling framework makes it possible to take general-purpose instrumentation code and use it online with the Full-Duplication framework with practically no modification.

Another important design choice of the instrumentation sampling framework is to ensure scalability for multiple processor machines. To trigger samples, the Full-Duplication framework relies on *counter-based checks*, which decrement and check a counter on all method entries and backedges. To ensure scalability on multi-processor machines, our implementation uses a separate counter for each processor, thus avoiding any bottlenecks or race conditions that occur when multiple thread simultaneously access a single counter.

5 Example Instrumentation: Edge Counters

Our instrumentation infrastructure was designed to allow incorporating a wide variety of instrumentations. The first instrumentation included in our system is intraprocedural edge counters [8]. The goal of edge counters is to collect the execution frequencies of the intraprocedural control flow edges between basic blocks. The execution frequencies of basic block can be derived easily from edge counts. Edge counters were chosen as the first instrumentation because they are useful for a variety of optimiations. They have been used *offline* in previous work for optimizations such as code reordering [27], instruction scheduling [19] and other classic code optimizations [12].

[2] The estimated speedup of all optimization levels (including FDO) is based on averages from offline performance measurements.

[3] The Jikes RVM adaptive system has a notion time that is based on the number time-based samples that have occured. This same notion of time is used by the alarm clock mechanism.

Previous work [8] has shown that careful placement of counters can reduce the overhead of collecting edge counts. However, one of the main advantage of instrumentation sampling is that it essentially eliminates the need to worry about the execution overhead of instrumentation. Therefore, to avoid unnecessary complexity, our system uses a simple counter placement and relies on the instrumentation sampling infrastructure to reduce the execution overhead. Optimal placement could still be used to reduce the counter space and code space, but is not necessary for the purpose of reducing time overhead.

Our simple strategy is to instrument all conditional branches to record the number of times they were taken and not taken. For each conditional branch, a counter is placed before the branch (to record the total number of times the branch was executed) and along the fallthrough path (to record the not-taken frequency). The taken-frequency is the difference between these counts.[4] Given this branch profile, execution frequencies for all control flow edges and basic blocks can be computed easily.

6 Feedback-Directed Optimizations

A wide variety of feedback-directed optimizations can benefit from edge count information. Our system currently implements the following four feedback-directed optimizations based on edge counts: splitting, code motion, method inlining, and loop unrolling. Each is described in the sections that follow.

6.1 Feedback-Directed Splitting

Splitting [10, 11] is a decade old compiler transformation originally designed to reduce the overhead of message sends in the Self programming language. The goal of splitting is to expose optimization opportunities by specializing sections of code within a method. Specialization is achieved by performing tail-duplication of conditional control flow to eliminating control flow *merges* (or side entrances) where data flow information would have been lost.

The main limitation of splitting is that the potential space increase is exponential. Algorithms that use simple static heuristics to perform aggressive splitting have been explored, such as *eager splitting* [10, 11], but were shown to cause excessive code expansion making them unusable in practice. Improved approaches, such as *reluctant splitting* [10, 11], delay splitting until later in the compilation process when dataflow information identifies regions of code where splitting would be beneficial. Although more effective than eager splitting, this approach requires substantial compiler infrastructure and can still result in unacceptable code expansion.

Our approach is to guide splitting efforts using profiling information; doing so offers two advantages. First, splitting efforts can be focused on hot paths, allowing more aggressive splitting than would be possible with static heuristics such as eager splitting, because space is not wasted duplicating cold code. Second, no modifications are necessary to the optimizing compiler other than the splitting transformation itself (unlike reluctant splitting). By performing splitting early in the compilation process, optimization opportunities are automatically exposed to the optimizations that follow.

The splitting algorithm, described in detail in Figure 2, attempts to split (and thus specialize) all heavily executed

Initialize:
```
1. Initialize hotMergeNodes: an ordered set
   containing all merge nodes, sorted by
   execution frequency.
2. methodSize = getOriginalMethodSize();
3. maxMethodSize = methodSize * EXPANSION_FACTOR;
4. hotMerge = hotMergeNodes.getHottestNode();
```
Iterate:
```
5. while (methodSize < maxMethodSize &&
6.        hotMerge.frequency() > MIN_SPLIT_THRESH)
7.    dupNode = hotMerge.clone();
8.    spaceExpansion += dupNode.getSize();
9.    Redirect the hottest incoming edge of
      hotMerge to jump to dupNode.
10.   Update edge frequencies.
11.   Update hotMergeNodes:  Insert new merge nodes
      and re-sort nodes with modified frequencies.
12.   hotMerge = hotMergeNodes.getHottestNode();
```

Figure 2: feedback-directed splitting algorithm.

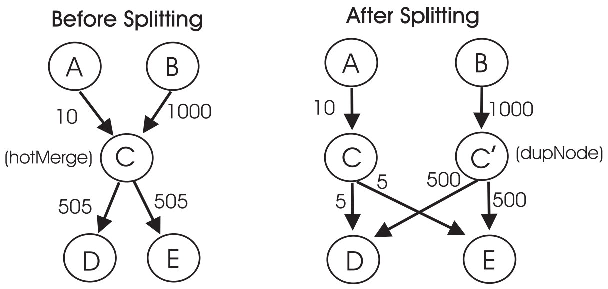

Figure 3: One iteration of feedback-directed splitting, where node `C` is the `hotMerge` node and `C'` is `dupNode`. After splitting, the incoming edge frequency of `C` and `C'` is divided among their outgoing edges according to the ratio of the original outgoing edges of `C` (in this case, 50–50).

paths through the method, while avoiding duplication of infrequently executed code. It is a simple greedy algorithm that eliminates heavily executed control flow merges by duplicating the merge node and redirecting the hottest incoming edge to jump to the duplicated (and now specialized) node. Loop cloning [10, 11] is not performed, so a *merge node* is defined to be any basic block that has multiple incoming control flow edges but is *not* a loop header.

Figure 3 shows an example of one iteration of the algorithm. Notice that after one merge node is spit, two new merge nodes are created. The algorithm iterates, continuing to greedily split the hottest merge node until either 1) a space expansion bound (`EXPANSION_FACTOR`) is reached for the method, or 2) there are no more merge nodes that are above a minimum splitting execution threshold (`MIN_SPLIT_THRESH`).
Step 10 is the only non-trivial step of the algorithm. After an edge is redirected from `hotMerge` to `dup` (step 9), several edge frequencies must be adjusted. If `hotMerge` has multiple outgoing edges (as shown in Figure 3) their frequencies cannot be known precisely because path profiling information is not available. Therefore, the weights of outgoing edges are approximated using a *constant ratios* assumption [20]. The ratio of the outgoing edge frequencies is assumed to stay the same as the original outgoing ratio (in this case, 50–50). Using this ratio, the total incoming frequency of `hotMerge` and `dup` is distributed among their outgoing edges.

[4] Counting total executions and using it to compute the taken-frequency resulted in slightly less disturbance to the the control flow than inserting an edge counter to explicitly count the taken frequency.

6.2 Feedback-Directed Method Inlining

Using profiling to improve method inlining decisions can have a significant impact on the performance of object-oriented languages [4, 17]. Jikes RVM has the ability to perform inlining based on static heuristics, as well adaptive inlining based on a call-edge profile collected via time-based sampling [3]. Although Jikes RVM's adaptive inlining shows performance improvements for many benchmarks, the sampled profile is fairly coarse grained and cannot be relied upon completely for all inlining decisions; therefore, static inlining heuristics are still used to make decisions for call sites that do not appear among the sampled edges.

Edge counts provide a finer granularity of profile information that can be used to fine tune inlining decisions. In our modified Jikes RVM system, call sites that are known to be infrequently executed are not inlined, even when the original static heuristics would have suggested inlining it.[5] Similarly, call sites that are frequently executed, but for some reason do not appear in the sampled profile, are given higher priority for inlining by expanding Jikes RVM's inlining size restrictions while processing those sites (and while processing any new code introduced by inlining at those sites).

Using edge counts in this way also has advantage of adding context-sensitivity to the inlining decisions [17]. All code inlined into the instrumented method (possibly through multiple levels of inlining) is monitored *in the context of its inlined position in that method*. It may turn out that certain inlining decisions can be changed for this method even though they were beneficial in other contexts.

6.3 Feedback-Directed Code Positioning

Code positioning rearranges the ordering of the basic blocks with the goal of 1) increasing locality, thus decreasing the number of instruction cache misses, and 2) reducing the number of unconditional branches executed. The original Jikes RVM system performs code reordering based on static heuristics. Our modified version uses edge counts to drive the *top-down* code positioning algorithm described by Pettis and Hansen [27]. The code reordering phase is one of the last phases of the machine independent optimizations.

6.4 Feedback-Directed Loop Unrolling

Loop unrolling is a transformation where the body of a loop is duplicated several times, allowing better optimization within the loop. The disadvantage is that loop unrolling increase space substantially if not applied selectively. The base version of Jikes RVM unrolls small (less than 100 instructions) inner loops 4 times. Our implementation uses edge counts to guide loop unrolling so that hot loops can be unrolled more while compile time and space is not wasted unrolling infrequently executed loops. Our current implementation uses a simple heuristic to increase, or decrease, the loop unrolling factor based on the execution frequency of the loop header node; the loop unroll factor is doubled for hot loops, and halved for cold loops.

7 Experimental Evaluation

This section presents an experimental evaluation of our implementation of the techniques described in this paper. As discussed in Section 4, our implementation uses the Jikes RVM; although not a fully complete JVM, the performance of Jikes RVM has been shown to be competitive with that of commercial JVMs. Our modified system is compared against the original Jikes RVM augmented with a more efficient implementation of guarded inlining [6]. This modified version offers higher performance than the original Jikes RVM, thus the baseline for all of our comparisons is the best performing configuration of our version of Jikes RVM. (Note: we intend to move our implementation into the open source version once the work is accepted for publication.)

Our experiments are divided into two sections. The first section describes a set of controlled experiments using the SPECjvm98 [29] benchmark suite, while the second section presents performance improvements using more realistic server benchmarks. All experiments were performed on a 500 MHz 6 processor IBM RS/6000 Model S80 with 6 GB of RAM running IBM AIX 4.3.2. For all benchmarks, Jikes RVM was run using 2 processors and a 400 MB heap.

7.1 SPECjvm98 Methodology

Understanding the performance impact of online instrumentation and FDO can be difficult because of the number of factors that ultimately affect performance, including the overhead of instrumentation, the effectiveness of the feedback-directed optimizations, and even the behavior of the underlying adaptive optimization system. To gain a solid understanding of the performance impact (both overhead and improvement) of our online approach, long-running versions of the SPECjvm98 benchmarks were used. A standard "autorun" execution of the SPECjvm98 benchmarks consists of 'n' executions of each benchmark in the same JVM; for our experiments, 'n' was chosen separately for each benchmark to allow all benchmarks to run for approximately 4 minutes (using the size 100 inputs). This experimental setup ensured that each benchmark executed long enough for the underlying adaptive optimization system to approach a steady state, and also allowed the performance of each individual execution to be monitored, making it easier to identify where overhead was incurred and performance was gained. The first column of Table 1 shows the number of iterations executed for each benchmark, and the second column shows the best time achieved using the underlying adaptive optimization system (without our FDO).

7.1.1 Steady state FDO performance

Figure 4 characterizes the peak performance impact of the feedback-directed optimizations described in Section 6. Each benchmark was run with and without our online approach for instrumentation and FDO. Figure 4 compares the best run from each scenario.[6] The height of each bar represents the percent improvement achieved by performing instrumentation and FDO. `_227_mtrt` resulted in the largest improvement of 20.9% while `_213_javac` showed the smallest improvement of 3.2%. The average improvement was 8.3%.

The patterns within each bar show how much each individual optimization contributes to the total performance win. The breakdown was computed by turning on each optimization, one at a time, starting with code reordering and working upward. This breakdown should be considered only a rough estimate of the individual contributions because there are many (mostly positive) interactions between the optimizations. Only one particular breakdown ordering was evaluated because the combined effect is what matters most.

[5] This does not include calls to small methods where inlining would reduce code size; such calls are always inlined.

[6] Recall that the base Jikes RVM system already performs loop unrolling, static and adaptive inlining, and code reordering. This graph shows the performance improvement of modifying these optimizations to take advantage of edge count information.

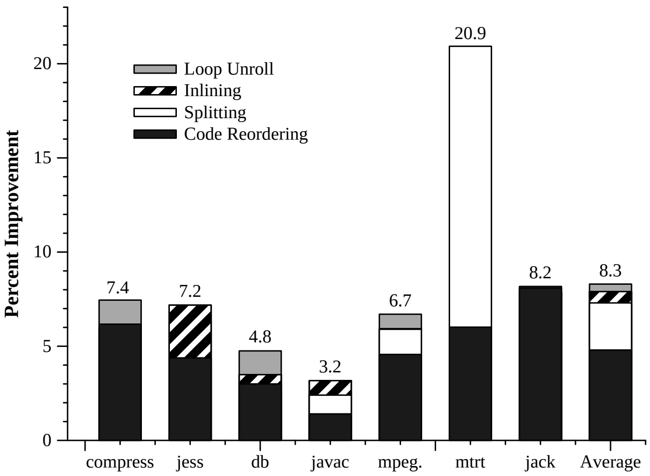

Figure 4: Peak performance improvement when using instrumentation and feedback-directed optimization.

Code reordering provided fairly consistent speedups for most benchmarks, averaging 6.1%. Splitting provides a large performance boost for _227_mtrt, which makes heavy use accessor methods, many of which are inlined using a runtime guard [14]. After splitting, many of these guards can be identified as redundant and eliminated.

Using edge counts to improve inlining decisions showed improvement for _202_jess, even with the base system already performing adaptive inlining, demonstrating that the fine-grained profiling provided by instrumentation can be used to improve coarse grained information. Loop unrolling provided little speedup for these benchmarks, showing only slight improvements for _201_compress and _209_db.

7.1.2 Online Performance

To measure online adaptive performance, previous work has often reported simple metrics, such as the performance of the first and best runs [3, 28]. Unfortunately these simple metrics do not tell the whole story regarding online performance because they disregard the performance of several runs, many of which may have incurred severe overhead. One alternative is to compare total execution time, rather than best time, however total time is somewhat arbitrary because short periods of high overhead can go unnoticed if the benchmark is configured to run long enough.

To convey the online performance of our technique, Figure 5 graphically compares the performance (with and without FDO) of all individual executions of the benchmarks. Each benchmark was run N times using the original adaptive system, and N times using online FDO. Each individual run (n) from the FDO runs was compared against the corresponding run n from the non-FDO runs to create a point in the graph.[7] The percent improvement due to FDO is the y-coordinate of each point (higher means more improvement due to FDO), and the total execution time (of the non-FDO run) is the x-coordinate, showing the point in time at which this particular improvement was achieved. Points from the same benchmark are connected by a line in the graph for easier viewing, but these lines have no formal significance.[8]

[7] To help reduce noise and make the graph more readable, benchmarks with individual execution times of less than 10 seconds (_227_mtrt and _202_jess) had consecutive timings grouped into pairs of two; the sum of each pair was then used as a single timing.

[8] Note to reviewers: For historic reasons, Jikes RVM guards all compilation with a global runtime lock that prevents any class loading from occuring, potentially causing all application threads to halt during compilation. This coarse grained locking mechanism, which is currently in the process of being removed, unnecessarily exaggerates the overhead of compilation. Because our online strategy assumes the ability to perform legitimate background compilation, our compilation thread was modified not to use this lock; although technically unsafe, it did not negatively affect any of our executions. We do not believe this to be an unrealistic limitation because we we expect to obtain these same results once the locking mechanism is fixed. Without this change, only one benchmarks (javac) showed increased overhead because the application thread slept during compilation.

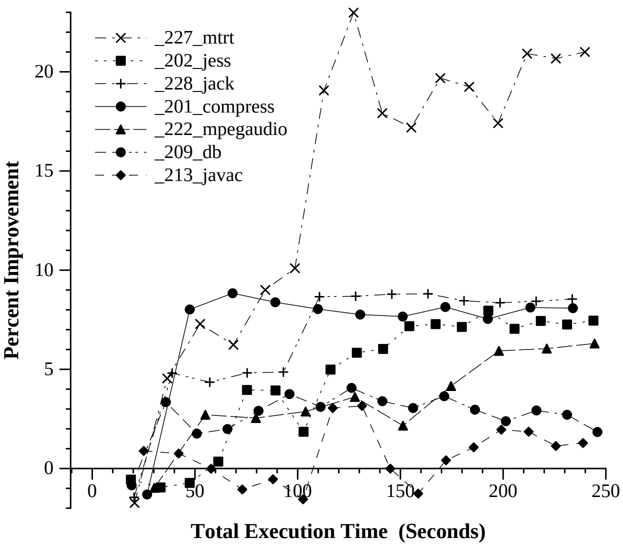

Figure 5: Online performance improvement when using instrumentation and feedback-directed optimizations.

Figure 5 shows that FDO improved the performance of most of the executions. As the benchmarks run longer, FDO continues to improve for most of benchmarks. Just as importantly, the largest degradation was less than 2%, demonstrating that the eventual peak performance does not come at the cost of severe overhead to any one execution. Additionally, the underlying adaptive system is non-deterministic and variations of a few percent are common, so slowdowns in the range of ∼2% (as there are several for _213_javac) could easily be noise as opposed to legitimate degradation.

These results are encouraging, particularly considering that much of the overhead being incurred is fixed cost overhead (such as the Full-Duplication compile time increase and runtime overhead) that will not increase as more instrumentation and optimizations are added to the system.

Note to the reviewers: Figure 5 shows a trend of slight degradation of the initial runs of each benchmark. We are currently experimenting with small adjustments to the model to further improve the balance between startup and peak performance.

7.1.3 Space Overhead

One concern of our instrumentation and FDO approach is the increase in space due to extra compilation. Additionally, our instrumentation uses the Full-Duplication sampling technique which doubles the size of the instrumented method, and 3 of our 4 feedback-directed optimizations have the potential to increase code size as well.

Fortunately, few methods needed to be instrumented to achieve the previously reported speedups. Columns 4–10 of Table 1 show the compilation statistics when FDO is performed. The first of these columns shows the total number of all method compilations that occured; the next 5 columns report the percentage of compilations performed by each

	Application Characteristics		Compilation Statistics (with FDO)						Space Overhead
			Total #	Percent Breakdown					
Benchmarks	# Runs	Best time	compilations	Base	O0	O1	O2	INST/FDO	% Increase
201_compress	11	20.5	382	93	2	3	1	1	6.3
202_jess	29	8.3	915	84	6	6	2	1	6.2
209_db	16	15.1	399	94	2	2	1	1	5.8
213_javac	16	13.2	1,575	70	16	32	1	1	4.6
222_mpegaudio	10	23.3	704	75	11	10	3	2	6.9
227_mtrt	49	4.4	634	78	8	10	2	1	6.6
228_jack	13	17.5	738	80	10	6	3	1	6.5
Average	21	14.6	763	82	8	7	2	1	6.1

Table 1: Recompilation statistics and space overhead for the SPECjvm98 benchmarks

	Throughput		
Point	Original	FDO	% Improvement
1	3,584.93	2,584.95	0.0 %
2	6,722.66	6,787.86	1.0 %
3	6,551.05	6,807.30	3.9 %
4	6,320.13	6,536.24	3.4 %
5	6,016.38	6,255.21	4.0 %
6	5,563.54	5,842.69	5.0 %
7	4,850.68	5,282.78	8.9 %
8	4,329.23	4,619.77	6.7 %

Table 2: SPECjbb2000 server benchmark performance, with and without online feedback-directed optimization

compiler and optimization level.[9] For all benchmarks, instrumentation and FDO was performed on only 1 or 2% of the compilations that occured.

The final column of Table 1 shows the total space increase when using instrumentation and FDO. Jikes RVM does not currently garbage collect methods that are no longer being executed, so space usages is computed by summing the machine code size of all compiled methods. The space increase ranges from 4.6% to 6.9%, and averaged 6.1%. This increase is reasonable given the speedups obtained, and is small compared to the space savings that could obtained using other techniques, such as adding an interpreter to the system [28], or by garbage collecting dead methods.

7.2 Server Benchmark Performance

Although the SPECjvm98 benchmarks were useful for understanding the online behavior of our approach, the real goal of this work is to improve the performance of server applications. This section evaluates the performance of our approach using the SPECjbb2000 server benchmark [29], a Java application designed to evaluate server performance by emulating a 3-tier middleware system.[10]

A standard compliance run of SPECjbb2000 was used, which involves executing a series of 8 "points" in succession; each point executes a 30 second warm-up period followed by 2 minutes of timed execution. Execution begins with a single thread (during point 1) and an additional thread is added for each additional point. The performance of each timed segment is reported as a throughput. Table 2 shows the performance of SPECjbb2000 both with and without FDO.[11] The performance improvement steadily increases through the first 7 points, reaching a maximum improvement of 8.9%, and declines slightly for the 8th point.

8 Additional Related Work

The instrumentation sampling technique used in this work was previously described in [5]. This prior work focused on the design and implementation of the sampling technique itself, and was evaluated in a completely offline setting. No online strategies using instrumentation sampling were presented or evaluated, and no feedback-directed optimizations were described. Our current work is a complete system that uses fully automatic, online feedback-directed optimization to significantly improve the performance of Java programs; the previous work on instrumentation sampling is just a small piece of the complete system described here.

Dynamo [7] is a binary translator that uses a technique called *next executing tail* (*NET*) [15] to approximate hot paths; their experimental results argue that traditional path instrumentation is ineffective for online use. Although this may be true given the nature of Dynamo, our system's use of edge/path profiling (as well as the systems described in 2.2) is fundamentally different. Our online approach assumes an execution environment where a lighter weight mechanism is used to identify hot sections of code and promotes them to higher levels of optimization; fine-grained profiling information is used only to improve the decisions made by the optimizing compiler. Dynamo, however, uses hot path information not only for performing code reordering, but also for identifying code that needs to move from interpreted to optimized state; therefore, Dynamo benefits most from profiling techniques that are quick, rather than accurate.

Traub et al. [30] describes *ephemeral instrumentation*, a technique that uses code patching to reduce the overhead of collecting an edge profiles. When used to guide superblock scheduling, the edge profiles collected by this technique produced speedups similar to those from a perfect profile, although a few benchmarks showed substantial differences. As discussed in Section 2.2, it is not clear whether this technique can be used to perform general instrumentation.

Our feedback-directed splitting is similar to superblock scheduling [12] which eliminates side entrances along hot paths by performing tail-duplication. The key difference is that superblock scheduling performs tail-duplication along a single hot *trace*, whereas our hot-path splitting allows duplication of multiple hot paths through a method.

Previous work has used dataflow information combined with offline profiling to guide code duplication. Ammons and Larus [2] used path profiles to guide code duplication for improving dataflow analysis. Bodik et al. [9] uses edge or

[9] All executing methods are first compiled with the baseline compiler, and may be subsequently compiled (possibly multiple times) with the optimizing compiler.

[10] Additional server benchmarks are currently being collected.

[11] For SPECjbb2000, Jikes RVM was run with adaptive inlining disabled because it substantially degrades performance, both with and without FDO. Once the underlying adaptive inlining mechanism is debugged for this benchmark, it will be re-incorporated.

path profiles to guide code restructuring for partial redundancy elimination and code motion. Our feedback-directed splitting differs from these approaches because it is not specific to one particular analysis or optimization, and is a simple greedy algorithm that is effective for online use.

9 Conclusions and Future Work

This paper describes the design, implementation, and evaluation of a fully automatic online approach for performing instrumentation and feedback directed optimization. Our approach uses instrumentation sampling to reduce the overhead of instrumentation, allowing a wide variety of traditionally offline instrumentation techniques to be performed online with little or no modification. As validation of our approach, one example instrumentation, intraprocedural edge counts, is presented. Several feedback-directed optimizations based on this instrumentation are also presented, including a novel profile-guided splitting algorithm.

Our experimental results show that with four relatively simple feedback-directed optimizations, peak performance can be substantially improved; speedups ranged from 3.2% to 20.9% (averaging 8.3%) for SPECjvm98 and 8.9% for SPECjbb2000. The overhead of our technique is also encouraging. No individual execution was degraded more than 2%, and much of this overhead is fixed cost overhead that will not increase as more instrumentation and feedback-directed optimizations are added to the system.

Our future plans are to use the online infrastructure presented here to explore additional instrumentation techniques and feedback-directed optimizations. Current possibilities include method-level specialization based on value profiles, as well as data locality optimizations based on memory reference profiles. We also plan to move our implementation into the open source release of Jikes RVM as well as continue exploring additional server applications.

10 Acknowledements

I would like to thank IBM Research and the entire Jikes RVM team for providing the infrastructure to make this research possible. I would also like to thank Stephen Fink, David Grove, Michael Hind, and Peter Sweeney for all of the helpful discussions about adaptive optimization. I am grateful to Martin Trap for supplying the loop unrolling implementation, as well as Stephen Fink, David Grove, Michael Hind for providing valuable feedback on an earlier draft of this paper. Finally, I thank Barbara Ryder and Michael Hind for their support of this work.

References

[1] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 72–84, 1998.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[4] M. Arnold, S. Fink, V. Sarkar, and P. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[5] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 168–179, Salt Lake City, Utah, 20–22 June 2001.

[6] M. Arnold and B. G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In submission, 2002.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[8] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[9] R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, 1998.

[10] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Comiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992.

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

[12] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using profile information to assist classic code optimizations. *Software—Practice and Experience*, 21(12):1301–1321, Dec. 1991.

[13] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[14] D. Detlefs and O. Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming*, 1999.

[15] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

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

[17] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Oct. 1995.

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

[19] W. Hwu et al. The superblock: An effective technique for VLIW and superscalar compilation. In *The Journal of Supercomputing*, pages 229–248. Kluwer Academic Publishers, 1993.

[20] O. Kaser and C. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24:55–72, 1998.

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

[22] M. Leone and P. Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):1–5, Sept. 1998.

[23] R. Marlet, C. Consel, and P. Boinot. Efficient incremental runtime specialization for free. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 281–292, 1999.

[24] M. Mock, C. Chambers, and S. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the 33th International Symposium on Microarchitecture, pp. 291–302*, Dec. 2000.

[25] T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.

[26] M. Paleczny, C. Vic, and C. Click. The Java Hotspot(tm) server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, 2001.

[27] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, White Plains, New York, 20–22 June 1990. *SIGPLAN Notices* 25(6), June 1990.

[28] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.

[29] The Standard Performance Evaluation Corporation. http://www.spec.org/.

[30] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 1999.

[31] J. Whaley. Partial method compilation using dynamic profile information. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.