Towards future method hotness prediction for Virtual Machines

Manjiri A. Namjoshi

Submitted to the Department of Electrical Engineering &
Computer Science and the Faculty of the Graduate School
of the University of Kansas in partial fulfillment of
the requirements for the degree of Master's of Science

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Perry Alexander

Dr. Andy Gill

Date Defended

2009/11/30

The Thesis Committee for Manjiri A. Namjoshi certifies
That this is the approved version of the following thesis:

Towards future method hotness prediction for Virtual Machines

Committee:

Chairperson

Date Approved

Acknowledgements

I thank Dr. Prasad Kulkarni, my advisor for his valuable guidance and inputs all through my thesis. Working with him has been a wonderful productive experience. He always had answers and definite direction at any point during the development of this project. I would also like to thank Dr. Perry Alexander and Dr. Andy Gill for being on my thesis committee and reviewing this thesis document. I would like to thank the department of Electrical Engineering and Computer Science at The University of Kansas for all its support.

I would like to thank my parents for their unconditional love and affection. My mother has been a source of inspiration in important phases of my life.

I thank all my friends here in Lawrence and in India for the fun and support I have had all my life.

Abstract

Application *profiling* is a popular technique that attempts to understand program behavior to improve its performance. *Offline* profiling, although beneficial for several applications, fails in cases where prior program runs may not be feasible, or if changes in input cause the profile to not match the behavior of the actual program run. Managed languages, like Java and C#, provide a unique opportunity to overcome the drawbacks of offline profiling by generating the profile information *online* during the current program run. Indeed, online profiling is extensively used in current VMs, especially during *selective compilation* to improve program *startup* performance, as well as during other feedback-directed optimizations.

In this thesis we illustrate the drawbacks of the current *reactive* mechanism of online profiling during selective compilation. Current VM profiling mechanisms are slow – thereby delaying associated transformations, and estimate future behavior based on the program's immediate past – leading to potential misspeculation that limit the benefits of compilation. We show that these drawbacks produce an average performance loss of over 14.5% on our set of benchmark programs, over an *ideal offline* approach that accurately compiles the hot methods early. We then propose and evaluate the potential of a novel strategy to achieve similar performance benefits with an online profiling approach. Our new online profiling strategy uses early determination of loop iteration bounds to predict future method hotness. We explore and present promising results on the potential, feasibility, and other issues involved for the successful implementation of this approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Application profiling is a technique of gathering information during the execution of a program. This information, or profile data, is used to monitor dynamic behavior of the application which can then be utilized to make compilation and optimization decisions, and improve application performance. Based on the tendencies observed during program's past and present runs if the program's execution is altered to improve performance then such a technique is called *feedback-directed optimization*. Feedback-directed optimization can be achieved using offline or an online profiling or combination of both. *Offline strategy* uses the profiling information collected from the previous runs and optimization takes place *after* program execution. *Online strategy* collects the profiling data dynamically, *during* the program execution itself. Profile-based compilation and optimization, when feasible and successful, can result in significant performance benefits.

In the earliest systems, feedback-directed optimization was performed using offline profiling [8, 24, 30, 32]. Profiling data is gathered by running an application for one or more times and used to summarize the program's behavior. These statistics are then used to optimize the application according to the predicted behavior. Since the optimization decisions are based on the predictions made using the profile data collected during previous program runs, it is clearly a *reactive* profiling technique. As a result, although offline profiling results in significant performance improvement, it fails in situations where (1)it is impractical to collect a profile prior to execution, or (2)a change in the execution environment or the input causes the application's behavior to differ from its behavior during the profiling run.

Virtual machines provide an environment for running the managed language code. For example, high level languages like Java [17] and Microsoft C# [10] are the managed languages that run on the Java Virtual Machine and CLR virtual machine respectively. The managed runtime environment (i.e., a Virtual Machine) presents a unique opportunity of performing profiling and optimizations *online* to overcome the previously mentioned drawbacks of offline profiling [2, 4, 5, 20]. The Java Virtual Machine implements online profiling to determine the program methods that take up most of the execution time, so that such methods can be compiled and optimized. Thus, based on the program behavior, compilation and optimization decisions are made adaptively. Such adaptive optimizations are critical to the startup as well as overall performance of managed runtime application. Even though the online profiling shows significant performance improvement by exploiting the runtime information, it still has some limitations:

1. Compilation and optimization decisions are based on the immediate past behavior of the application which may not remain the same in the future. Although profiling is performed online, it still relies on the stale information and assumes that the behavior will not undergo any change in future. Thus, it is again a *reactive* profiling mechanism and can result in performance

degradation in case of incorrect future prediction.

2. The *compilation and optimization decisions are delayed* because of the lack of adequate profiling information during the program startup. Thus, the application suffers a long startup delay.

In this thesis, we quantify the performance degradation due to drawbacks of the current adaptive online profiling technique and propose a new online profiling mechanism for virtual machines. Our new profiling technique tries to exploit the ability to determine the values of certain program variables during execution. The knowledge of exact values of particular variables at some specific points during execution run can help not only to predict, but accurately determine the significant characteristics of the program behavior. This detailed program monitoring, possible due to the managed runtime environments, can help drive the compilation decisions earlier as well as avoid incorrect future speculations. Most current virtual machines employ a policy of *selective compilation*. Since, virtual machine performs compilation dynamically, it introduces compilation overhead at runtime. Selective compilation uses profiling information to focus compilation resources only on the frequently executed code portions to minimize this overhead [3, 28]. Usually, individual methods are these code portions and *method hotness* is determined by the proportion of method's execution period in the total application's execution time. Method hotness is a loosely defined term that indicates the impact of a method's individual performance on the overall execution time of the application. In our new profiling technique, we use the exact values of *loop iteration bounds*, available at runtime, to accurately predict the future invocation frequency of all the methods called from that loop, prior to the method's first invocation. Thus, our profiling technique can remove both the drawbacks of current profiling. Due to the accurate future prediction, incorrect speculations can be reduced and due to early detection of hot methods the compilation decisions need not be delayed. We use classfile annotations for Java bytecodes in order to mark instructions that can fetch the loop iteration bounds. The loops are detected and the annotations are inserted statically. Also, the virtual machine is modified to parse these annotations and perform new actions at the marked instructions. Our experimental results demonstrate that our new profiling technique can significantly improve application *startup* performance by accurately compiling hot methods early. The specific contributions of this thesis are as follows:

1. measure and manifest the performance degradation due to the drawbacks of the current profiling strategy.

2. suggest a new profiling technique that aims at overcoming the mentioned drawbacks by collecting information regarding crucial sections of code, to predict the future accurately.

3. evaluate potential of complete as well as practical knowledge of loop iteration bounds to predict future method hotness.

4. study issues regarding the feasibility and runtime cost of our new profiling strategy for selective compilation.

The rest of this thesis is organized as follows. In chapter 2, we describe work related to the areas of online profiling and selective compilation. We outline our experimental framework in Chapter 3. In Chapter 4 we demonstrate the drawbacks of current implementations of online profiling. We propose and evaluate our new online profiling technique in Chapters 5 and 6. We then describe the results of our technique with only practically analyzable loops in chapter 7. Further we outline our plan for the future maturation of the online profiling framework in chapter 8, and finally draw our conclusions in chapter 9.

Chapter 2

Background

The managed runtime languages offer some very significant advantages over statically compiled languages, like portability, safety guarantees, ability to install software components at runtime due to dynamic class loading, etc. Due to these requirements, managed languages cannot use the traditional static program optimizations. Thus, it becomes challenging to achieve high performance. In response to this limitation, adaptive optimizations are used extensively to improve performance by relying on the online profiles to drive runtime optimization decisions. In this chapter, we provide background information and related work in the areas of selective compilation, online profiling techniques and reducing the overhead of dynamic compilation using selective optimization and binary file annotations.

2.1 Selective Compilation

Interpreters were probably the first widely used virtual machines. But, the execution of interpreted code is much slower than executing native and optimized code. To improve interpreter performance, virtual machines employ runtime compilers, commonly known as Just-in-Time (JIT) compiler [12]. JIT compilers perform compilations and optimizations of code sequences to native code at runtime. Since the compilation time is now added in the total execution time, it is important to perform compilations judiciously. The virtual machines thus aim at reducing this compilation overhead.

Initially, JIT compilers dynamically compiled each new method upon its first invocation. This simple strategy is called *lazy compilation* [28]. Since it only compiles methods on-demand, it avoided compilation of the methods that are never invoked during the application execution. However, this strategy is still found to compile several methods that are either never reached again or invoked so infrequently as to not justify the time and resources spent in compiling the code. In order to further reduce compilation overhead, virtual machines exploit the well-known fact that most programs spend most of their time in a small part of the code [3, 6, 26]. Hence, only frequently executed code sequences or "hot spots" can be compiled. This technique is called *Selective compilation* [11, 18, 19, 35]. Using this technique, mostly the current virtual machines use "cheap" implementation of interpretation by default and a more expensive compilation for only the program hot spots. Some virtual machines may use a fast non-optimizing compiler for all the methods and only optimize the frequently executed methods, thus introducing the different levels of optimizations [2, 9]. Selective compilation technique needs a profiling mechanism which can detect or identify program hot spots to drive the compilation decisions. The identified hot code sequences may then further qualify for higher levels of optimizations as well. Profiling is also used during feedback-directed optimizations to specialize, revert, or adapt optimization decisions based on changing program conditions [4, 32].

2.2 Online Profiling for Selective Compilation

For effective online feedback-directed optimizations, it is necessary for a virtual machine to implement a profiling technique that can collect accurate information with low overhead. Current VMs use one or a combination of different profiling techniques that are based on *counters* [19, 22, 31] and *sampling* [2, 18]. The approach based on counters, updates counts for method entry points and loop backward branches. Both these counters are associated with every method. If the method invocations and/or loop backward branch counts exceed a fixed threshold then the method is selected for compilation. The second approach based on sampling, uses interrupts generated by the hardware performance monitors. The system is interrupted to record the method(s) on top of the stack. If the same method is encountered multiple times such that it exceeds a predetermined sampling threshold, then it can be queued for compilation.

Both the currently used online profiling techniques by the VMs, are reactive in nature. They collect information about the immediate past program execution and make the optimization decisions regarding the future program behavior. Current techniques fail to priori predict the future and assume that the future behavior will remain similar to the past behavior. But, previous works show that the program behavior variability requires the systems to adapt according to future rather than recent past behavior [14]. The work shows that the programs exhibit significant behavior variations and thus the reactive mechanism is bound to misspeculate future. Since the compilations and optimizations are performed online, incorrect decisions can result in performance loss. Incorrect speculations can adversely affect the application's startup performance and short-running programs. Another drawback of the current profiling techniques is delay introduced in the compilation decisions due to the insufficient profile data. The longer an application is profiled, the future prediction is further delayed and thus lower will be the reuse of the predictions [13]. Sampling using an external clock trigger distributes the profiling overhead over a longer time interval than corresponding counter-based schemes, thereby, potentially delaying the compilation decisions even more. Additionally, sampling-based profiling also introduces non-determinism that can complicate performance analysis and system debugging. Instead, we propose a new profiling mechanism that predicts future application behavior, early, and as much or more deterministically than current profiling techniques.

A few research works have also explored the use of hardware performance monitors to reduce the overhead of collecting profiles [1, 7]. In addition to harboring the same drawbacks of current counter and sampling-based profiling schemes, systems using hardware performance monitors have often also found it difficult to associate the very low-level information obtained via hardware monitors with the higher-level program elements that actually influence the counters. One earlier work demonstrated that delaying important compilation decisions can be harmful to application startup performance [29]. However, this prior work only focused on reducing the delay between the detection and compilation of hot methods, and still uses and suffers from all the drawbacks of current profiling-based schemes that our research attempts to solve.

2.3 Static Analysis to Aid Dynamic Optimization

Some earlier research works used static analysis of the source code and annotations to carry information concerning compiler optimizations. These works aimed at making the bytecodes more expressive in conveying compiler optimizations to the JIT. The annotation aware JIT system uses this information to generate high-performance native code [23, 27, 33]. Hummel et al. used an annotation-based framework to send static information about register allocation in the Kaffe JIT. Their scheme improved the performance of register allocation at the cost of greatly increased code-size for the intermediate *classfile*. Also, some works employed static analysis to reduce JIT compiler's compilation overhead. Annotations consists of analysis information collected off-line which is used to speedup compilations and optimizations that are performed dynamically [27]. Krintz et al. predicted hot methods for selective compilation based on profile information based on previous application runs to reduce compilation overhead [33]. Pominville demonstrated the framework using annotations for elimination of need to generate native code for array bounds and null pointer checks. The static analysis guaranteed that these checks are not needed and thus optimized the classfiles. In our proposed technique, static analysis is used to detect loops and identify loop entry, exit instructions. Annotations are used to convey offsets of the bytecode instructions corresponding to program points where values of loop iteration bound variables can be collected at runtime.

Chapter 3

Experimental Framework

3.1 Benchmarks

We used the Standard Performance Evaluation Corporation's (SPEC) JVM98 benchmarks to conduct our experiments [34]. Table 3.1 lists the benchmarks belonging to this suite. There are two input sizes small(10) and large(100), which is indicated by the suffix of each benchmark's name. Next columns give total number of methods executed, total (application+library) and only application methods detected hot by the HotSpot VM and number of loops in application classes.

3.2 HotSpot VM

We conducted all our experiments using **Java HotSpot virtual machine** (build 1.7.0-ea-b24) [31], Sun Microsystem's high performance VM.

Name	Methods Executed	Hot Methods Total	Hot Methods App.	App Loops
_201_compress_10	1410	21	17	26
_201_compress_100	1410	22	18	
_202_jess_10	1741	41	22	171
_202_jess_100	1757	80	47	
_205_raytrace_10	1515	75	49	43
_205_raytrace_100	1516	104	77	
_209_db_10	1415	36	11	21
_209_db_100	1418	39	9	
_213_javac_10	2135	89	42	237
_213_javac_100	2173	409	308	
_222_mpegaudio_10	1574	55	50	77
_222_mpegaudio_100	1576	99	77	
_227_mtrt_10	1524	78	52	43
_227_mtrt_100	1531	106	79	
_228_jack_10	1652	107	20	89
_228_jack_100	1656	172	70	

Table 3.1. Benchmarks used in our experiments

3.2.1 HotSpot Components

This section gives a brief overview of the Java HotSpot virtual machine, and provides a short description of the components that are directly relevant to this thesis.

The Java Standard Edition Platform contains two implementations of the Java VM, the Client and the Server. For all our experiments we use the client VM because it has been specially tuned to reduce application startup time and memory footprint, making it particularly well-suited for client environments.

Adaptive compiler: In order to overcome the drawbacks of JIT compilation, HotSpot employs an adaptive compilation and optimization technique. The JVM initially launches an application using the interpreter, and analyzes the code, as it runs, to detect the critical hot spots in the program. The online profiler does a counter based sampling and collects counts for each method at method entry and backward branches. The HotSpot VM has a statically set, default *CompileThreshold* of **1500**. When the sum of method invocation count and loop backedge count exceeds this threshold then the method is detected as hot and it is queued for compilation. Thus, every method is individually profiled, compiled and optimized. Hotspot's optimizing compiler does not profile the method any further once it is optimized.

Template Interpreter: It is a default interpreter in the HotSpot VM. Everytime the VM initializes, the interpreter is generated at runtime from assembler templates. These templates are translated into native language code that runs as an interpreter within HotSpot. Advantage of the template interpreter is that machine code does the dispatching of each bytecode. But, the interpreter itself is quite complicated.

Classfile Parser: Every classfile is loaded dynamically and parsed through the VM's classfile parser. While the classfile is parsed, the method structures are created corresponding to each method present in the classfile. To maintain the portability of classfiles, all the non-java attributes (or annotations) are ignored by the classfile parser.

3.2.2 Our Experimental Conditions

On-stack replacement: By default, the HotSpot VM employs On-stack replacement (OSR) strategy. If the method is detected to be hot due to long-running loops then special version of the method is compiled. The interpretation can stop after this compiled code is available and execution of compiled method begins by allowing entry in the middle of the loop. Due to the complexity of the OSR process [15,21], OSR compilation, in most cases, is only supported in commercial VMs and some other extensive research projects such as Jikes RVM [2]. Additionally, a fair comparison between the default HotSpot compilation policy (with OSR) and our early compilation strategy will require substantial updates to enable the HotSpot VM to employ the generated native code during the same method invocation or loop iteration as when the method is first detected hot and compiled. Currently, the compiled code is only employed in the invocation/iteration after the hotness detection. We have disabled the OSR compilation in HotSpot to keep our study simple and to allow more straight-forward analysis of our results.

Background compilation: The HotSpot VM spawns a separate compiler thread in order to carry out compilation of methods. Thus, even though a method is queued, it may not be immediately compiled by the compiler thread. Hence, we turn off the background compilation to make sure that all the methods are compiled immediately after they are queued. With no background compilation, the compiler thread blocks the application, or the main thread, compiles the method and then the application execution resumes. This helps us measure the benefits of early compilation more deterministically.

DelayCompileInstall: Along with the detection of hot methods, the counter-based sampling technique is used to carry out other optimizations. The early compiled code does not get a chance to undergo these optimizations. Thus, in order to measure the benefit of early compilation solely, we do not allow the default technique to perform optimizations on methods that are profiled for longer period. Thus, even for the default runs, we compile the marked hot methods during their first invocations. But, the VM starts the execution of the native code only after the method's total count reaches the *CompileThreshold*.

3.2.3 Threshold Selection

The default *CompileThreshold* of the HotSpot VM is 1500. But, we experimentally choose the threshold that gives the best performance on an average for the SPEC JVM98 benchmarks. The optimal value of the hotness threshold is very critical for the startup performance of the VM. Too low value of the threshold may result in detection of too many methods to be hot thus increasing the compilation overhead, due to the compilation of unimportant methods, resulting in the performance degradation. Or, too high threshold may not detect some methods to be hot and avoid compilation resulting in the performance degradation due to large number of interpretations. To find the correct threshold values to use for our results, we experimented with several different thresholds to find the one that achieves the best performance for our experimental conditions and benchmark set. Please note that this approach is also the current state-of-the-art

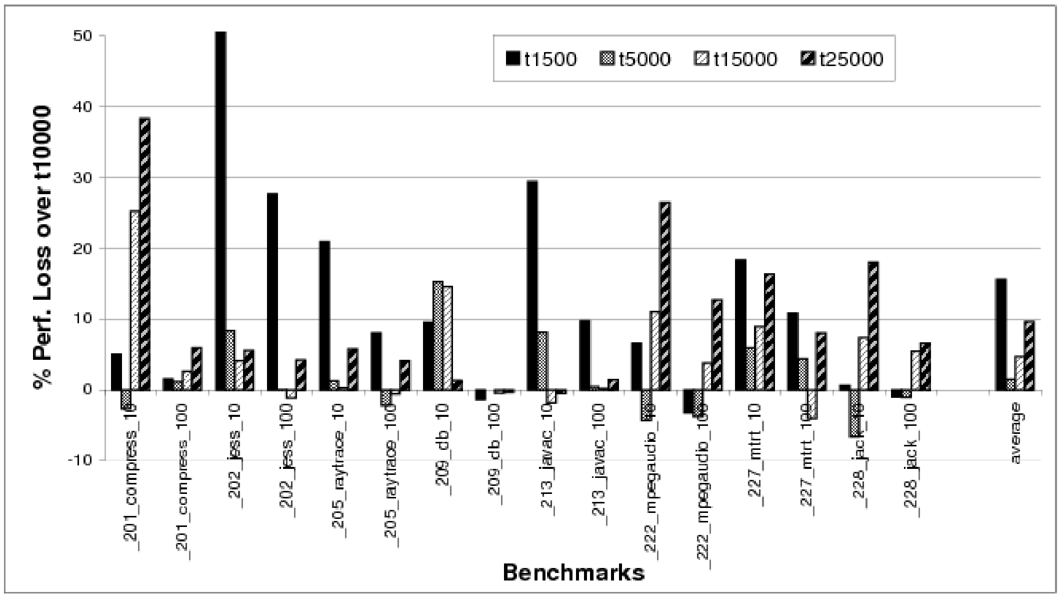


Figure 3.1. Comparison of benchmark performances at different hotness thresholds with the threshold of 10,000

method for detecting hotness thresholds to use in production JVMs. The results of our experiment for a subset of the tested thresholds (1500, 5000, 10000, 15000, 25000) are presented in Figure 3.1. In this figure, each benchmark's performance at the threshold of 10,000 is used as the base to compare its performance at the remaining thresholds. The results indicate that, although individual benchmark performances at the various thresholds may vary, a compile threshold of about 10,000 achieves the best performance result, on average. Therefore, we selected the threshold of 10,000 as the base for our experiments in this work.

3.3 Soot

In order to conduct some of our experiments, we require to indicate some additional profiling information to the VM in the form of classfile annotations. Thus, we extended Soot, a Java Optimization and Annotation framework, to insert method level attributes. Also, we used soot to statically detect and analyze loops. The method attributes are inserted to indicate the instruction offsets of loop entries and exits. Finally, all our experiments were performed on a single core Intel(R) Xeon(TM) 2.80GHz processor using Fedora linux 7 as the operating system. All the performance results report the median over 10 runs for each benchmark-configuration pair.

Chapter 4

Drawbacks of Current Online Profiling

As discussed earlier, the JIT compiler dynamically compiles all the methods, since the execution of the compiled native code is much faster than the interpretation. But, compiling all the methods incurs a heavy compilation overhead resulting in performance degradation since the compiler runs in the application's execution time. The JIT compilation technique resulted in significant delay in the startup of a program and affected more for the short running application. Thus, future generation VM employed selective compilation technique using online profiling. It has been known that programs spend most of the time in a small fraction of code, 90% of the execution time comes from 10% of the code. In order to reduce the compilation overhead and taking advantage that most programs spend vast majority of time in small part of code, selective compilation only compiles frequently executed sections of the code. Online profiling strategy helps to detect the hot sections of code, which is carried out during the execution of the application. However, the current online profiling approach has some limitations. There are two main drawbacks, namely:

1. incorrect future prediction, since the past behavior may undergo a change in future.
2. delayed compilation decision, due to the lack of sufficient profiling information.

In this chapter, we quantify the performance impacts due to each of the above mentioned drawbacks.

4.1 Incorrect Future Prediction

The selective compilation with the online profiling strategy is a reactive mechanism to detect hot methods. The counter based sampling keeps a record of the number of times methods have been invoked as well as the number of back branches taken, in the past. It assumes that the profiled program behavior will continue to be the same in the future. Based on this assumption, if the counts exceed the threshold then that method is detected hot. But, there is no guarantee about the method's future invocation or loop branches. Thus, if the future speculation goes wrong then in worse case the VM will incur the compilation cost to compile such a method and not get a chance to execute the compiled code even once.

In order to determine the incorrectly speculated methods, for our benchmarks we consider the theory of choosing the correct compile threshold. The theory to determine the correct compile threshold for the selective compilation technique, aims at reducing the worst-case damage of online compilation, in case when the future speculation goes wrong. This approach is based on the *ski-renting princi-*

ple [16,25]. In our context, ski renting principle will try to make a choice between continuing interpretation or compilation of a method, which will reduce the repeated interpretation cost. This principle aims to reduce the worst case damage, when we have no knowledge of the future behavior. According to this principle, every method is interpreted for some number of times before it is compiled, in order to ensure that it is worth introducing the compilation overhead if the method continues to execute. The worse-case is when a compiled method never gets a chance to execute. The ski-renting principle gives the worse-case overhead of twice the amount of time spent in compilation. Any other algorithm will give a greater worse-case damage. Let Y be the time taken for interpretation of a method and X be the time required to execute the compiled method. If a method is executed n times, then the compilation of this method can be beneficial only if:

$$nX + compilationoverhead < nY \tag{4.1}$$

If we apply the ski-renting principle, then let m be the number of times the method is already interpreted before compilation, then :

$$mX + compilationoverhead = mY \tag{4.2}$$

In the above equation, m is nothing but the optimal compile threshold that gives the best performance. We have stated in the earlier chapter that empirically the best performance is achieved on an average for the threshold of 10000 for the Spec JVM98 benchmarks.

We employ the ski-renting principle and Equation 4.2 conservatively to say that a method is incorrectly speculated to be hot if $C < I$, where the method is interpreted I number of times before compilation and for C number of times the compiled method is executed. Thus, in our case, if a method that is detected hot and compiled at a hotness threshold of 10,000 has a total count (invocation + backedge) of less than 20,000, then we consider the detection as a case of incorrect speculation. We calculated the number of incorrectly speculated methods for all our benchmarks. These results are presented in Figure 4.1. It shows that on an average 23% of methods are incorrectly sent for compilation. Also, incorrect speculation can be as high as 53% for some benchmarks.

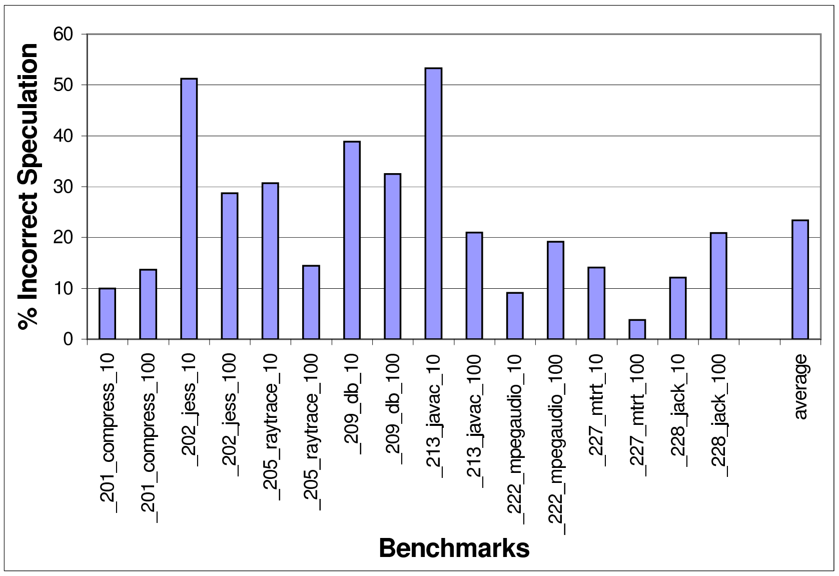


Figure 4.1. Percentage of methods incorrectly speculated as hot

Further, we measured the performance improvement if the wrongly speculated methods are prevented from compilation. Method is incorrectly speculated hot when the benefit due to execution of the optimized native code over pure interpretation is not sufficient enough to offset the compilation overhead. Indeed, avoiding the compilation of such incorrectly speculated hot methods, resulted in a overall 3.79% increase in average performance.

4.2 Delayed Compilation Decisions

The second important drawback of the current online profiling technique is that the compilation decisions are delayed due to the lack of sufficient profiling information. Thus, according to equation 4.2, the VM has to interpret methods for m number of times. Then, if the method counters exceed this threshold, the method will get queued for compilation. For the HotSpot VM, as mentioned in the previous chapter, the value of m is set to be 10000. We believe that interpreting each method for such a large value can be quite detrimental to the startup performance of applications. Due to the delayed compilation decision, it is also possible that the overall benefit of compilation is reduced as well. To empirically quantify the actual performance loss caused due to this drawback, we performed the following steps:

1. We collected histograms for each benchmark by running all the benchmarks as a prior profiling run. The histograms give the sum of the method invocation count and backedge count for each method in a benchmark. This sum is denoted as *total count*. Since, this total count for each method quantifies the total execution time spent in a particular method, it is a direct measure of the method hotness. In HotSpot VM, a method is detected as *hot* if its *total count* >= 10000.

2. Next, we employed the Soot annotation framework to insert method level attribute in the hot methods detected in the previous step.

3. Finally, we modified the HotSpot VM to recognize this annotation. We inserted a flag field to indicate if the method is hot or cold. This flag is set for all the annotated methods and only such methods are allowed to

be compiled. Our technique of *early* compilation sends hot methods for compilation right during their first invocation. While, the *default* queues methods for compilation after their total counts exceed or match 10000. Thus, we try to record the maximum early compilation benefit that we can get by queuing the hot methods for compilation at the count of 1 instead of 10000.

Figure 4.2 shows the performance gain due to compiling hot methods *early*, at the hotness threshold of 1 compared to compiling them *normal*, at 10000. Thus, early compilation of hot methods results in a performance benefit of 14.5%, on average over the default technique. As we can observe the benefits for smaller size

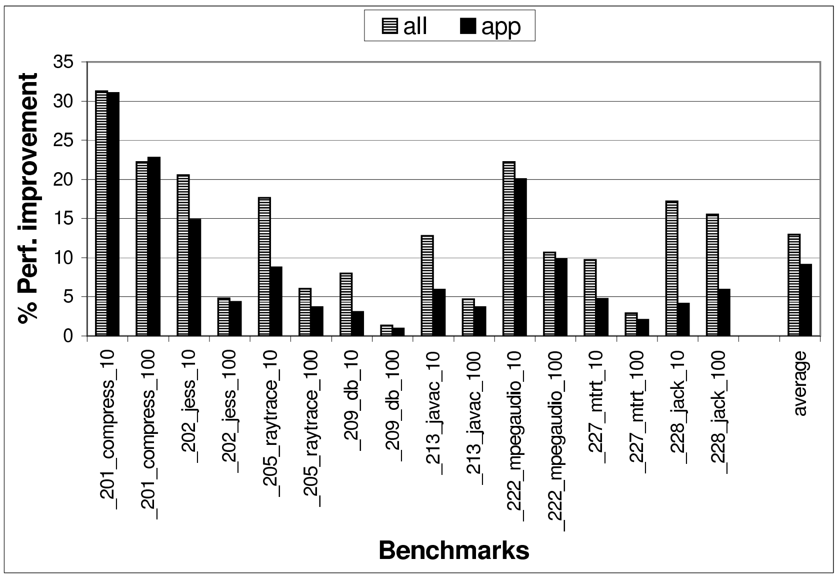


Figure 4.2. *Ideal* benefits of early compilation of hots methods

(input size 10) benchmarks are much better (17.7%) than for the longer running or the larger size (input size 100) benchmarks (11.4%). This is because we are impacting only the startup performance. That is, with the longer execution time of a benchmark, the early benefit gets distributed over a larger period and thus percentage improvement measured get reduced. Also, since the early technique is compiling methods at the count of 1, there are no incorrect speculations.

Thus, we have shown in this section that drawbacks of current online profiling technique causes some performance loss. Certainly, if the current technique's limitations can be removed or reduced then significant performance improvement is possible. We have empirically shown by the percentage of incorrectly speculated methods that it is important to find about method's future hotness than its past total counts. Similarly, early compilation decisions can improve performance by utilizing the maximum compilation benefit. Hence, we propose a new technique in order to priori predict the future hotness behavior. Our new approach can reduce both the current approaches drawbacks.

4.3 Implementation Challenges

One of the greatest challenge in this work was to understand the various parts of the extensive source code of the HotSpot VM. We needed to determine the internals of the interpreter implementation, the sections of code where profiling is performed on every method entry and loop back branches, how a method is queued for compilation, etc. We also explored the various runtime flags that helped us get log information about the benchmark run, like which methods are queued for compilation and also the method invocation and loop backedge counts at which the particular methods are queued. Further, the HotSpot VM interpreter is a performance critical component and its code is generated during the VM startup. During initialization, the VM generates the code pieces that the interpreter performs for each bytecode. The VM creates *assembler templates*, which contain the platform dependent machine code. This machine code implements the behavior of the bytecode. Thus, it was difficult to know the specific x86 instructions that will be generated by the particular assembler templates.

We had some issues due to the early compilation of library methods. Unless the VM is initialized, the adaptive compiler cannot compile any method. With early compilation, many library methods are invoked and thus detected hot even before the VM is initialized. The main thread calls the VM and tries to queue the library method, but fails and resets the method's counts. This causes an unnecessary overhead during an early compilation run. To avoid these failing attempts, we had to check if the VM is initialized before any method is queued for compilation. This problem never occurs for the application methods, because the application execution cannot begin unless the VM is initialized.

Chapter 5

Loop Iteration Counts for Early Prediction of Method Hotness

In the previous chapter we demonstrated that early compilation of hot methods can significantly benefit the startup performance of applications. In order to be able to early compile hot methods it is essential to determine the future method hotness. Thus, in this chapter we evaluate the possibility of using the values of loop iteration bounds in-order to calculate future invocation frequency of the methods invoked from loops and the amount of time that will be spent for execution of a method with a loop. It has been known that loops constitute the most executed sections of applications and thus are best candidates to apply profiling. It is unlikely for a method to be invoked for a large number of times unless it is called from at least one loop or the method itself is recursive. It is generally easier to analyze loops than recursive methods. Also, most programs have much more number of loops than recursive methods and thus to make it simpler we have restricted our scope of current work to only analyze loop-based behavior. Thus, it is reasonable to assume that any method will be hot only if it is called from loops and/or itself contains a long running loop. With the long running loop present in a method, a significant amount of time might be spent in execution of that method. Experimentally, we evaluate the potential of knowing accurate loop bounds to detect hot methods early. Since our profiling technique is online, we can get the exact values of loop iteration bounds for most of the loops from the managed runtime environment even before entering a loop. In further chapters we will give statistics of the number of loops for which it is possible to get the loop bounds before entering the loop for each benchmark. Initially, we evaluate the detection of hot methods early if we know accurate loop bounds of all the methods before loop is entered. Then, we shall evaluate the ability to detect hot methods early by considering only the loops for which it is feasible to know their loop bounds early.

Here, we conceptually describe with the help of some simple programs (not the code snippets from any benchmark), the use of knowledge of loop iteration bounds to determine the future method hotness. Figure 5.1 shows program call-graph for a very simple program. For this particular program, loop iteration bounds is able to detect methods to be hot before or during their first invocation itself. The program execution begins with the `main` method, but the `main` method is called just once and does not contain any loop in its body. Program execution will not stay in `main` method for long and hence it is not a hot method in this example. Further, `hot_func1` method is hot because it contains a long running loop with a bound greater than the compile threshold. Method `hot_func2` is also hot. Since it is called from a hot loop, its future invocation count will exceed the compile threshold. In contrast, method `cold_func` is not hot even though it is called from a hot method, since it is neither called from a loop nor does it's body contain any loop.

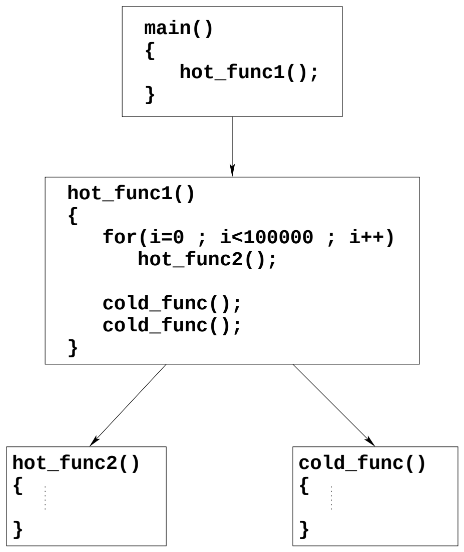
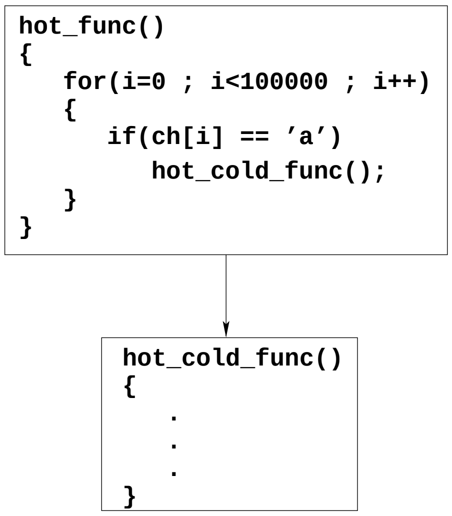


Figure 5.1. Simple partial program flowgraph

With this example, we have shown that for some cases it is possible to predict future method hotness very accurately just by loop bounds information. But, this precision may not always be possible. Next example shows that a method may or may not be hot even though it is called from a hot loop. Figure 5.2 shows a partial call-graph in which the method `hot_cold_func` may or may not be hot depending upon the input data. Thus, we expect such scenarios that depend on conditional program control-flow and values of program input to affect the accuracy of method hotness detection even with the knowledge of all loop iteration bounds.

5.1 Simulation Setup

In the previous section, we hypothesized that knowledge of the loop iteration bounds (just prior to the loop entry) may enable early determination of most hot methods. In this section we describe the experiments we performed to assess this



```
hot_func()
{
   for(i=0 ; i<100000 ; i++)
   {
      if(ch[i] == 'a')
         hot_cold_func();
   }
}
```

```
hot_cold_func()
{
   .
   .
   .
}
```

Figure 5.2. Partial program flowgraph with difficult to predict methods

hypothesis. To perform these experiments, we require information regarding all loop iteration bounds, as well as factors contributing towards method hotness, such as method invocation frequencies and backedge counts. To limit the complexity of our simulation setup, we have restricted the scope of this study to only the methods and loops present in the application classfiles (ignoring library methods and loops). This restriction is purely for easing the clarity and feasibility of our analysis, and in no way affects the generality of this work. We perform the following steps to generate a trace file that contains this information for later offline analysis.

1. We employ the Soot bytecode analysis and annotation framework [36] to detect the loops in every application method, and identify their corresponding loop entry and exit instructions. Figure 5.4 (a) shows an example Java program with three methods, *main*, *methodA*, and *methodB*. Figure 5.4 (b) (minus the annotation instructions shown in bold) presents the Java VM bytecode instructions generated for the Java program in Figure 5.4 (a). All

loop entry and exit instructions detected by Soot are indicated in a bold font.

2. We employ Soot again to annotate the application methods in our benchmark classfiles with *attributes* to identify the particular method as well as indicate all the loop entry and exit instructions to the VM. As mentioned earlier, JVM bytecode specification allows the addition of user-defined attributes to the Java classfiles. Unfortunately, the JVM bytecode specification does not provide any *direct* means of annotating individual instructions. Therefore, we use method-level attributes to forward the necessary instruction-level information to the VM. Figure 5.4 (b) shows the method-level annotations that we add to a Java classfile. The first annotation, labeled *MethodID*, is a unique integer identifying each application method. The next annotation, labeled *Loop_entry*, provides the *(loop entry instruction offset, loop_id)* pair for each loop in that method. Finally, the *Loop_exit* annotation specifies the *(loop exit instruction offset, loop_id)* pair for the loop identified by *loop_id*.

3. We then use a modified version of the HotSpot Java virtual machine to recognize our new attributes, and output appropriate trace data at corresponding points during the execution of the benchmark programs. On executing the annotated classfiles, the JVM prints out markers to identify method entry, all later iterations of a loop, the loop exit, and the number of iterations of the loop. Figure 5.4 (b).

4. Finally, the trace file is post-processed to shift the loop iteration bound, originally printed at the end of the loop, to the start of the loop, so as to

```
class TestProgram {

    public static void main(String args[]) {
        for(int i=0 ; i<2 ; i++) {
            methodA(10);
    }

    static void methodB() {
        System.out.print(" ");
    }
```

```
static void methodA(int len) {
    int i;
    while(i < len){
        if(i%2 != 0)
            methodB();
        i++;
    }
}
```

(a) Java Source Program

```
public static void main(java.lang.String[]);
   Code:
    Stack=2, Locals=2, Args_size=1
    0: iconst_0
    1: istore_1
    2: iload_1
    3: iconst_2
    4: if_icmpge 18
    7: bipush 10
    9: invokestatic #2  // call methodA
    12: iinc 1, 1
    15: goto 2
    18: return

   Annotation:
    MethodID = 1
    Loop_entry = #4,0
    Loop_exit = #18,0


static void methodB();
   Annotation:
    MethodID = 3
```

```
static void methodA(int);
   Code:
    Stack=2, Locals=2, Args_size=1
    0: iconst_0
    1: istore_1
    2: iload_1
    3: iload_0
    4: if icmpge 22
    7: iload_1
    8: iconst_2
    9: irem
   10: ifeq 16
   13: invokestatic #3  // call methodB
   16: iinc 1, 1
   19: goto 2
   22: return

   Annotation:
    MethodID = 2
    Loop_entry = #4,1
    Loop_exit = #22,1
```

(b) Annotated JVM Bytecode Classfile

```
f1 $0 f2 $1 , f3 , , f3 , , f3 , , f3 , , f3 , %1 10 ,
      f2 $1 , f3 , , f3 , , f3 , , f3 , , f3 , %1 10 ,
   %0 2
```

(c) Trace File Generated by the HotSpot JVM

```
f1 $0 2  f2 $110  , f3 , , f3 , , f3 , , f3 , , f3 , %1 ,
         f2 $110  , f3 , , f3 , , f3 , , f3 , , f3 , %1 ,
   %0
```

(d) Post−Processed Trace File

Figure 5.3. Process of generation of the trace file containing comprehensive information of all loop iteration bounds and method invocations for each benchmark

make it available to our offline analysis program on the first iteration of each loop. The trace file from Figure 5.4 (c) after post-processing appears as shown in Figure 5.4 (d). The shifted loop iteration bound is indicated in a bold font.

A loop may have more than one exit instruction, and Soot allows us to detect all loop entry and exit instructions correctly.

```
public class MultipleLoopExitExample {

    static void methodA() {
        for(int i=0;i<10;i++) {
          try{
            int j = 3;
            if(i%j==0) {
                break;
            }
          }
          catch(Exception e) {
              return;
          }
       }
    }
}

Annotation:
 MethodID = 1
 Loop_entry = #5,68
 Loop_exit = #23,68#30,68
```

```
static void methodA();
  Code:
    Stack=2, Locals=2, Args_size=0
    0:iconst_0
    1:istore_0
    2:iload_0
    3:bipush10
    5:if_icmpge30
    8:iconst_3
    9:istore_1
    10:iload_0
    11:iload_1
    12:irem
    13:ifne19
    16:goto30
    19:goto24
    22:astore_1
    23:return
    24:iinc0, 1
    27:goto2
    30:return
```

Figure 5.4. Example of multiple loop exit program

Program profile information from the trace file allows us to accurately calculate the number of method invocations and loop backedges, as well as the number of iterations for every loop. Such information is later used by our offline analyzer to estimate the earliest point when a method can be detected hot. We realize that a more direct approach of testing the potential of loop iteration bounds for early hotness detection would be to actually implement all the necessary modifications in a real virtual machine, like HotSpot, and compare the resulting runtime performances. However, the presented simulation-based strategy allows several advantages over this direct approach:

- The virtual machine is a complex piece of software. Indeed, a major portion of the HotSpot interpreter (the component where most of our changes would be localized) is code to dynamically generate a native language interpreter at runtime. Using offline simulations allows us an opportunity to confirm our hypothesis before attempting to overhaul the profiling and compilation mechanism in the HotSpot VM.

- Unless our basic, naive strategy works directly out of the box (which, as we later show is not the case), performing our experiments offline on the once-generated trace file allows us greater options of investigating alternative techniques and tuning our heuristics without having to modify the VM every time. With the flexibility now available to settle on the best technique and heuristics by performing offline experiments, we only need to port over those changes to the VM.

5.2 Tracefile Post Analysis Description

We use the procedure described in the last section to generate trace files for all our benchmark programs. Now, based on the upfront knowledge of all loop iteration bounds before loop entry, we, in this section, describe our algorithm to detect the hot methods early, along with results on the algorithm's detection accuracy. We also measure the performance benefit of compiling the detected hot methods at their earliest detection counts over the default reactive VM compilation strategy.

Our algorithm to analyze the trace file simulates the occurrence of *interesting* events in the exact order that they occur during the actual execution of the program. In addition, the trace file contains information about the exact number of loop iterations at the entry point of every loop. The interesting events required to detect method hotness include loop entry and exit, loop back-edges, and method invocations. The analysis program maintains and updates additional data structures to facilitate the simulation process. The most significant of these structures include:

loop_stack: A single structure that holds information regarding the loop identifier, total iteration bound, and the current iteration count for every active loop during program simulation. The structure is dynamically updated by pushing a new record onto the loop_stack on loop entry, and popping a record on loop exit.

method_info: A method-specific data structure that records the dynamic number of method invocations for each *loop context*. A loop context is defined by all the loops in the loop stack when that method is reached.

On the occurrence of each interesting event in the trace file, the analyzer takes the following steps to update the data structures and predict hot methods along with outputting the counts when they are first detected hot:

Loop Entry: The pattern `$<loop_id> <loop_bound>` indicates entry into a loop. The simulator pushes a new record on top of the loop_stack, along with its *total* loop bound.

Loop Backedge: A `','` in the trace file indicates the occurrence of a backedge to start the next iteration of the loop on top of the stack. The simulator increments the corresponding *current* loop iteration count.

Loop Exit: The record on top of the loop stack is popped on occurrence of the pattern `'%'` in the trace file.

Method Entry: The symbol `f<method_id>` indicates an invocation of the method denoted by its method_id, and the method's invocation count in all relevant loop contexts in the method_info structure is incremented. For the first invocation of a method in every new loop context, the simulator estimates the method's total count using the formula:

$$total\_cnt = \frac{(inv + back) * tot\_iter}{curr\_iter} \tag{5.1}$$

where,
inv is the current method invocation count,
back is the current method backedge count,
tot_iter is the total loop iteration bound, and
curr_iter is the current iteration count for the present loop context.
If $total\_cnt > compile\ threshold$, then this method will be sent for compilation. Thus, method compilation now happens when the method's total (past + future) count is estimated to exceed the VM's compilation threshold.

Figure 5.5 demonstrates the simulation of the trace file that is produced for the Java program in Figure 5.4. Assume for this example that the ***hotness threshold* is 10**, i.e., a method is marked hot and sent for compilation if its total count (given by Equation 5.1) exceeds 10. The processed trace file from Figure 5.4 (d) is reproduced in Figure 5.5 (a). Figures 5.5 (b)–(g) show the states of the simulator data structures, *loop_stack* and *method_info*, along with the counts at which the methods are detected hot for six simulation snapshots marked in Figure 5.5 (a). These events are described below:

1. Figure 5.5 (b) shows the state of the loop_stack after the simulation program reads in the symbols `$0`, corresponding to stage (1) in Figure 5.5 (a). Each application loop is provided with a unique identifier that is recorded in the

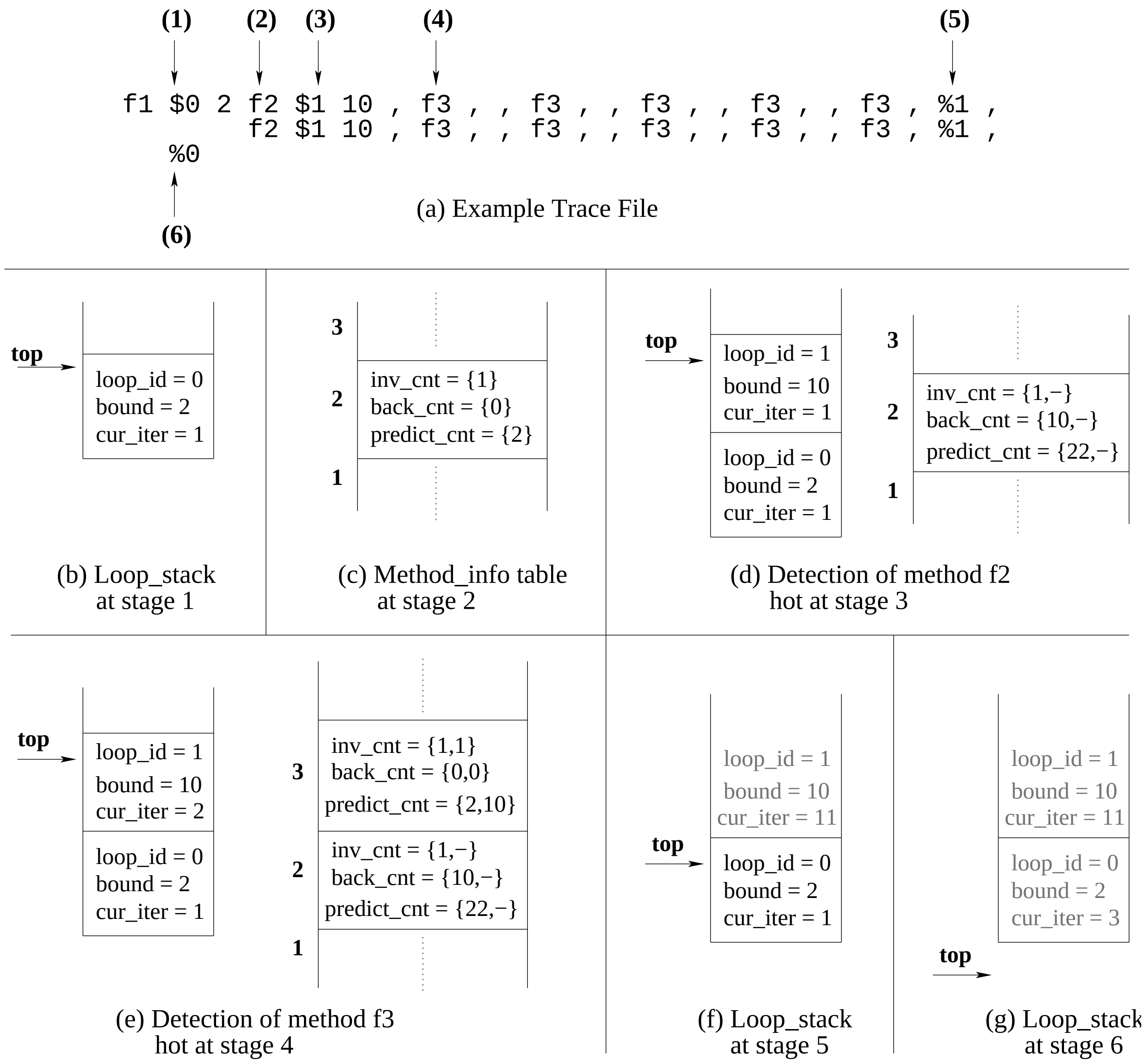


Figure 5.5. Demonstration of the simulation algorithm on an example trace file (for hotness threshold of 10)

field `loop_id`. The loop iteration bound is noted in the field `bound`. The field `curr_iter` indicates the current iteration number of the loop and is initialized to 1.

2. Figure 5.5 (c) shows how the method_info array is updated on reading the symbol `f2`. The fields `inv_cnt` and `back_cnt`, corresponding to the number

of invocations and backedges already seen for each method, are vectors with an entry for each loop level. Thus, if a method invocation or backedge is seen in an inner loop, then it is recorded as seen in all outer loop levels as well. These fields are set to 1 and 0 respectively at the loop level 0 on reading symbol `f2`. The simulator then employs Equation 5.1 to predict the method's total count for each current loop context. Method `f2` presently has only one loop context defined by the loop \$0. Method `f2`'s total count in this loop context is: $(1 + 0) * 2/1 = 2$, and is indicated in the field `predict_cnt`. The method is detected to not be hot.

3. Figure 5.5 (d) shows the state of the two simulator structures at stage (3) of the simulation, after having read the symbol `$1`. A new loop record corresponding to `loop_id`=1 is pushed onto the loop_stack. Since this loop exists in the method `f2`, its `back_cnt` is updated to the loop's iteration count of 10. Equation 5.1 is again used to calculate `f2`'s total count, given by: $(1 + 10) * 2/1 = 22$. Since `f2`'s total count is greater than the compile threshold (assumed as 10 in this example), `f2` will be sent for compilation at this stage. Thus, method `f2` is compiled during its first invocation, and the compiled version will be available before its next invocation. Therefore, the invocation count at which `f2` is first predicted to be hot is output as *1* by the analyzer.

4. The symbol `f3` is encountered in stage 4. Figure 5.5 (e) shows the updated states of the loop_stack and the method_info array. Note that we have already seen the first ',' symbol in the trace file, and accordingly updated the `curr_iter` field for the loop on top of the loop_stack. Method `f3` is present in two loop contexts, one defined by the loop nest formed by loops

$0 and $1, and the other defined by the loop $0. `f3`'s total count is calculated as $(1+0)*(10*2)/(2*1) = 10$ and $(1+0)*2/1 = 2$ for the loop contexts ($1_$0) and ($0) respectively. Since total count at context ($1_$0) is equal to the compile threshold, `f3` can be sent for compilation at this stage. Thus, `f3`'s first predicted hotness count is also output as *1*.

5. The loop record from loop_id=1 is popped off the loop_stack on encountering the symbol `%1` in stage 5. Please note that the *curr_iter* is equal to one more than the *bound* for loop_id=1 at this stage.

6. Finally, the loop record for loop_id=0 is popped off the loop_stack on reading symbol `%0` in stage 6 of Figure 5.5.

The analysis of each per-benchmark trace file outputs all detected hot methods along with the invocation counts when they are first detected hot. We again employ Soot to annotate the benchmark classfiles to indicate the hot methods, as well as their detected compilation counts to the VM. Our modified VM then compiles all annotated methods at their specified counts during benchmark execution. This framework allows us to evaluate the performance potential of using loop iteration bounds information for detecting and compiling hot methods early. Our present analysis framework only supports single-threaded application programs, and so we had to leave out the multi-threaded benchmark *_227_mtrt* from the results in this and future sections.

5.3 Results of Basic Post Analysis Algorithm

The analysis results are presented in Table 5.1 and Figures 5.7 and 5.8. The first column in Table 5.1 lists the benchmark name, along with its input size. The next set of two columns, labeled *Hot Methods* shows the *Actual* and *Predicted* number of hot methods during our simulation. *Actual* hot methods are the methods that are compiled during a normal run of the HotSpot VM. Our aim is to predict the hotness characteristic of the same methods early. The number of *predicted* hot methods is generally greater than their actual number due to the presence of *false positives* during our trace-file simulations. False positives are methods that are wrongly predicted hot by our algorithm, and indicate the inability of loop iteration bounds to provide accurate results by themselves. As mentioned earlier, instead of only relying on the predicted *future* method counts, our current implementation employs the sum of the past and future method counts to determine hot methods. Therefore, our current implementation has no *true negatives.* This means that all the *actual* hot methods, if not predicted to be hot early, will be sent for compilation at the method threshold count of 10,000 (although this happens very rarely in practice).

Benchmark	Hot Methods		False Positives
	Actual	Predicted	
_201_compress_10	17	19	2
_201_compress_100	18	22	4
_202_jess_10	22	25	3
_202_jess_100	47	51	4
_205_raytrace_10	49	67	18
_205_raytrace_100	71	78	7
_209_db_10	11	12	1
_209_db_100	9	14	5
_213_javac_10	42	100	58
_213_javac_100	309	527	218
_222_mpegaudio_10	50	159	109
_222_mpegaudio_100	77	169	92
_228_jack_10	20	63	43
_228_jack_100	69	120	51

Table 5.1. Results of Trace-File Simulation to Predict Hot Methods

Even for the correct detections, methods may be predicted hot at total counts greater than 1. Among other reasons, this unintentional delay may be caused due to inadequate future information available from loops with small iteration bounds. For example, in Figure 5.6, information available in `Loop1` regarding the future invocation of method `funcA` is insufficient to predict `funcA` hot ($(1+0)*8000/1 = 8,000$ by Equation 5.1 is less than compile threshold of 10,000). However, `funcA` will be detected hot in `Loop2`, although, after it has already been invoked for 8000 times.

```
          for(i=0 ; i<100 ; i++) {
LOOP1:         for(j=0 ; j<80 ; j++)
                   funcA();
LOOP2:         for(j=0 ; j<100 ; j++)
                   funcA();
          }
```

Figure 5.6. Small loop bounds delay the prediction of hot methods

Figure 5.7 ignores the false positives and shows the total (invocation + backedge) counts at which the *actual* hot methods are first predicted to be hot and compiled. Thus, this figure shows that knowledge of loop iteration bounds does allow most hot methods to be detected much earlier than the default compile threshold of 10,000. The presence of mostly recursive methods coupled with very few big loops explains the particularly harsh numbers for the benchmark *_202_jess*.

Figure 5.8 compares the performance of the various benchmark programs when using the simulation results to compile the detected methods early with the default system that compiles methods at hotness counts of 10,000. The results, for the most part, are not surprising and are intuitive. The large number of false positives (seen in Table 5.1) results in a significant performance degradation for several benchmarks, including *_213_javac*, *_222_mpegaudio* and *_228_jack*. The inability

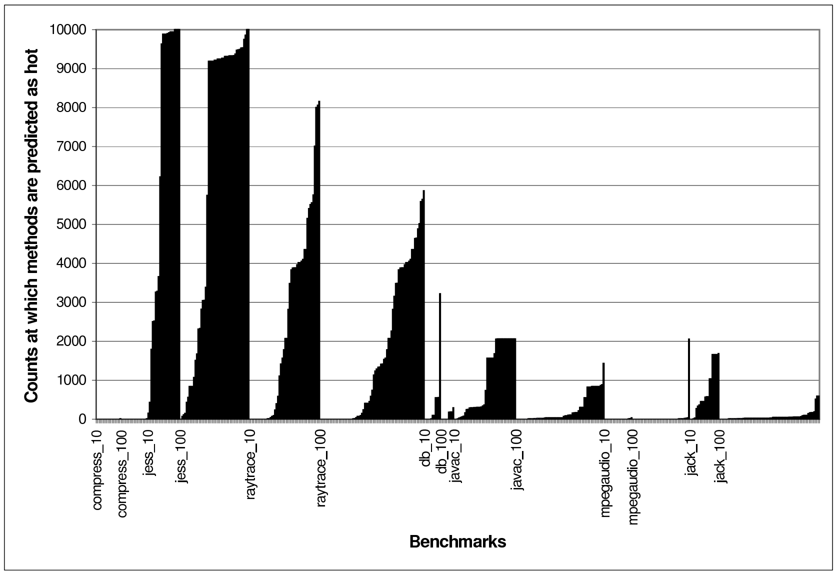


Figure 5.7. Total (invocation + backedge) counts when actual hot methods are first predicted hot

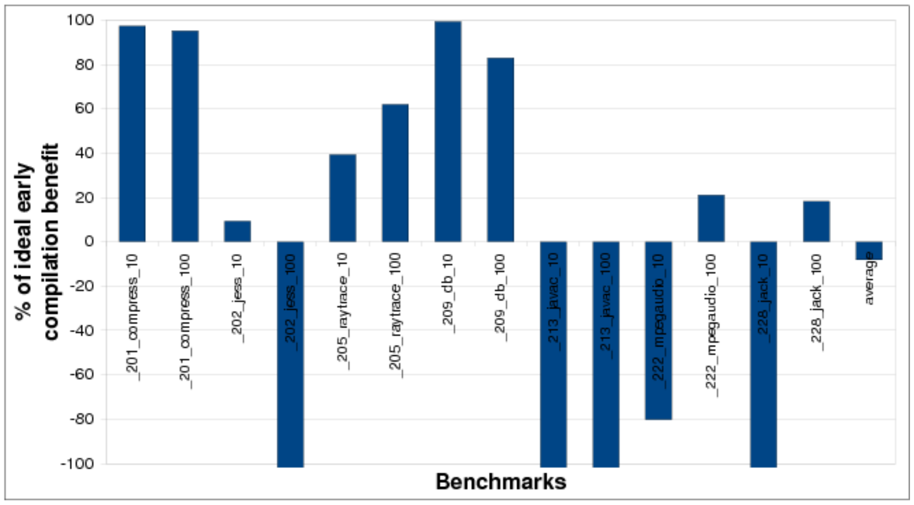


Figure 5.8. Benchmark performance when compiling methods as predicted by our analysis algorithm compared to the ideal offline performance from the *app* field in Figure 4.2

of our analysis framework to detect hot methods early, as seen for benchmarks *_202_jess* and *_205_raytrace* in Figure 5.7, also results in significantly reducing the gains compared to ideal early compilation. Finally, benchmarks with good prediction accuracies and early prediction counts, such as *_201_compress* and *_209_db*, are able to get most of the performance benefit of early compilation.

5.4 Implementation Challenges

We have extended Soot framework to detect loops, identify offsets corresponding to each loop entry and exit instructions, and to insert the method level annotations in the application classfiles. Though Soot is able to correctly identify offsets present in the original classfiles, it is unable to preserve these offsets in the classfiles that are generated. Soot converts bytecodes to an intermediate representation called Jimple and then back to bytecodes. This is not a 1:1 transformation. It also performs some optimizations on the input classfiles before it can generate valid bytecodes. This transformation and optimizations cause instruction offsets in the generated classfiles to change. Thus, the offset values inserted as annotations do not match the offsets corresponding to the newly generated classfiles. For this reason, we need to split this process into two separate steps.

1. First pass of basic Soot program will only transform all the classfiles. Then, we allow another Soot program which identifies loops and offsets to run on these transformed classfiles. This program outputs a file containing the assigned method identifiers, loop identifiers and their entry and exit offsets.

2. Our final Soot program runs on the original classfiles. It reads the offsets from the file which is generated in the previous step and inserts this information as method level attributes.

Chapter 6

Delaying Early Prediction to Improve Quality

From the last chapter, we see that early compilation based only on the knowledge of loop iteration bounds can produce significant number of false positives causing unnecessary compilation overhead, and a resulting loss in performance for several benchmarks. In this chapter, we analyze these results to discover the most common causes for the incorrect predictions, and present and evaluate a technique to improve the quality of our predictions.

6.1 Main causes of false predictions

The examples presented in Figure 6.1 indicate the main causes of false predictions (cold methods predicted as hot) during our early compilation results shown in the last chapter. Both the examples in Figure 6.1 show trace file fragments that are derived from actual benchmark programs. The first fragment derived from the benchmark program *_205_raytrace* explains the effect of variable inner

```
$7 11 ... $18 1395 f100 , f100 , ... , %18
          $18    1 f100 , %18
          $18    1 f100 , %18
          .
          .
          .
          $18    1 f100 , %18
%7
```

(a) Example 1 (_205_raytrace): Impact of Variable Loop Bounds

```
$3 503 ... $4 2048 f100 , , , , ... , %4
           $4 2048 f100 , , , , ... , %4
           $4 2048 f100 , , , , ... , %4
           .
           .
           .
           $4 2048 f100 , , , , ... , %4
%3
```

(b) Example 2 (_222_mpegaudio): Impact of Conditional Statemen

Figure 6.1. Impact of runtime variables on method hotness prediction

loop iteration bounds on method hotness detection. The loop nest consists of an outer loop (loop_id = 7) with an iteration bound of 11, and an inner loop (loop_id = 18) with a variable iteration bound. The inner loop iterates for 1395 times during the outer loop's first iteration, but only iterates once for all future outer loop iterations. The method with id=100 is invoked once during every iteration of the inner loop. The total invocation count of method f100 calculated during its first invocation $((1+0)*(11*1395)/(1*1) = 15,345)$ exceeds the threshold count of 10,000, and hence the method is sent for compilation during its first invocation itself. However, this prediction for f100 proves too optimistic due to loop $18's smaller future iteration counts.

The second example fragment presented in Figure 6.1 (b) explains the impact of conditional control-flow statements on the predicted future method invocation counts. The loop nest consists of two loops (id=3 and id=4) with fixed iteration counts of 503 and 2048 respectively. The conditional control-flow statements surrounding method f100 (not captured in the trace file) only allow its invocation during the first iteration of loop \$4. However, method f100 is predicted hot during its first invocation itself ($(1+0)*(503*2048)/(1*1) = 24,144$), which ultimately proves incorrect. For each false prediction, the cumulative gain due to faster execution of the native code fails to eclipse its compilation overhead, resulting in a net loss of application performance.

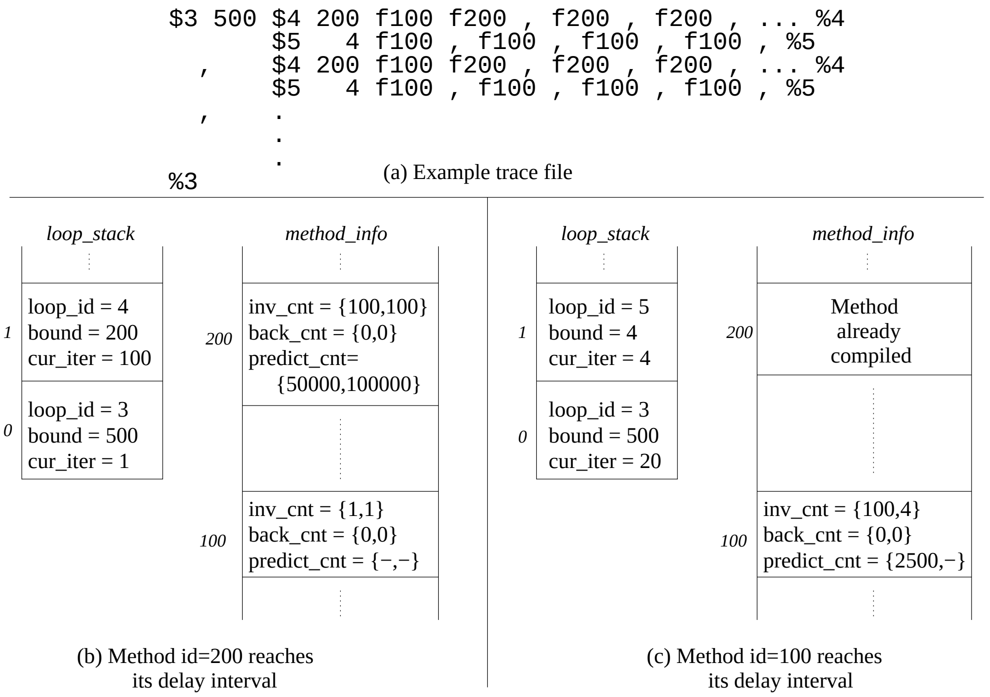
Thus, the above examples reveal the following causes for the hotness mispredictions:

- non-consistent inner loop iteration bounds, and
- conditional control-flow statements affecting method invocations.

6.2 Improved Tracefile Analysis

To reduce the number of false positives, we propose a new heuristic that avoids making a hotness prediction the first time a method is seen in any *loop context*, but delays the decision until sufficient *history* of the method counts is available for that method in that loop context. This heuristic is designed to address both the primary causes of false predictions witnessed during our simulation experiments.

Figure 6.2 (a) shows an example trace file to demonstrate the effect of delaying the prediction of hot methods. The example uses a *delay factor* of 1% of the default compile threshold of 10,000. Thus, for any loop context, a method is first checked for hotness only after that method's total count has already reached a 100 in that loop context. The trace file in Figure 6.2 (a) contains one loop at outer level 0 (id=3), and two inner loops at level 1 (id=4 and id=5) entered during each iteration of the outer loop. Method f100 is called only in the first iteration of loop \$4, and in all iterations of loop \$5; method f200 is invoked in all iterations of loop



```
$3 500 $4 200 f100 f200 , f200 , f200 , ... %4
       $5   4 f100 , f100 , f100 , f100 , %5
  ,    $4 200 f100 f200 , f200 , f200 , ... %4
       $5   4 f100 , f100 , f100 , f100 , %5
  ,    .
       .
       .
%3
```

Figure 6.2. Demonstration of the simulation algorithm with early compilation delayed for a *delay factor* of 1% (of 10,000)

$4, and never called from loop $5. Figure 6.2 (b) shows the states of the loop_stack and method_info when method f200's total count first reaches 100. Method f200's predicted count (by using Equation 5.1) is also shown for both loop contexts (just $4 and $3_$4). Since f200's predict_cnt is greater than 10,000 for at least one of these loop contexts, f200 can be sent for compilation at this stage. Method f100 reaches is delay interval (only for the loop context $3_$5) when the state of the loop_stack and method_info structures is as shown in Figure 6.2 (c). However, its predicted count (at context $3_$5), calculated as $((100 + 0) * 500/20 = 2,500)$, is less than the compile threshold, and so the method is predicted as cold at this point, and not sent for compilation. Note that without this delay factor, f100 would have been predicted hot during its first invocation itself. Thus, the delay factor allows us to eliminate some false positives, at the cost of postponing the compilation of *actual* hot methods by a similar factor.

The ideal delay factor varies for different methods. For example, although a delay factor of 1% is able to eliminate compilation of false positives for examples in Figures 6.2 (c) and 6.1 (b), a higher delay factor will be necessary to eliminate the false positive in Figure 6.1 (a).

6.3 Results of Improved Post Analysis Algorithm

Table 6.1 shows the number of false positives at various delay factors for the benchmarks in our set. While extremely small delay factors suffice to purge all the false positives for some benchmarks, such as *_201_compress* and *_209_db*, some very large delay factors are required for others, such as *_213_javac* and *_202_jess*. As noted earlier, a higher delay factor also suspends the early detection of *actual* hot methods longer, resulting in an erosion of the desired benefits due to early compilation. Thus, finding the ideal delay factor is important for achieving the gains of early compilation.

Figure 6.3 plots the ratio of the average performance improvement seen at various delay factors to the average ideal early compilation benefit available from Figure 4.2. The *best* delay factor is the benchmark-specific delay at which that benchmark achieves its best improvement. The *best* delay factor for each benchmark is indicated in Table 6.1 by expressing the appropriate false positive number in bold fonts. Figure 6.3 shows that, for constant delay factors, the performance improves rapidly with initial increases in delay factors and fluctuates (or decreases slightly) as prediction delays start affecting the benefit due to early compilation of actual hot methods. The best performance (8.5%, on average, over default VM compiling only *app* methods) is achieved for benchmark-specific delay factors,

Benchmark	Act. hot	False Positives at Delay Factors (% of 10,000)										
		0%	1%	3%	7%	10%	20%	30%	40%	50%	70%	90%
_201_compress_10	17	2	2	**0**	0	0	0	0	0	0	0	0
_201_compress_100	18	4	2	**1**	1	1	1	1	1	1	0	0
_202_jess_10	22	3	3	3	3	3	3	3	3	3	3	**0**
_202_jess_100	47	4	4	2	2	2	2	2	1	1	0	**0**
_205_raytrace_10	49	18	5	**4**	4	4	4	2	2	2	0	0
_205_raytrace_100	71	7	5	**4**	4	4	4	2	2	2	0	0
_209_db_10	11	**1**	1	1	0	0	0	0	0	0	0	0
_209_db_100	9	5	4	**2**	0	0	0	0	0	0	0	0
_213_javac_10	42	58	57	53	38	38	37	31	24	15	6	**0**
_213_javac_100	309	218	132	105	84	71	16	**1**	1	1	0	0
_222_mpegaudio_10	50	109	14	8	**5**	4	1	1	1	0	0	0
_222_mpegaudio_100	77	92	**4**	2	0	0	0	0	0	0	0	0
_228_jack_10	20	43	41	36	25	**10**	2	1	1	0	0	0
_228_jack_100	69	51	43	**14**	0	0	0	0	0	0	0	0

Table 6.1. Actual hot methods and false positives at different delay factors

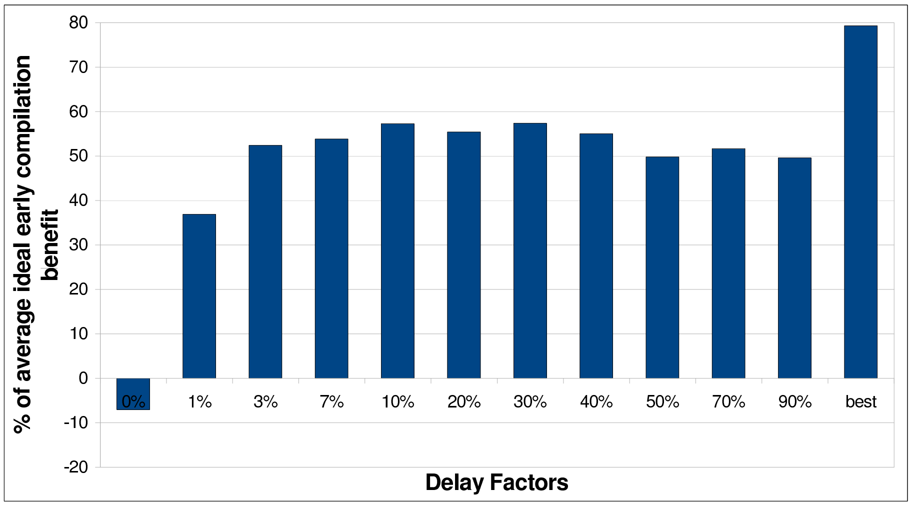


Figure 6.3. *Average* performance results of delaying compilation to improve prediction accuracy, as a percentage of ideal early compilation benefit at selected delay factors

indicating that there is some potential for tuning the delay factors as well.

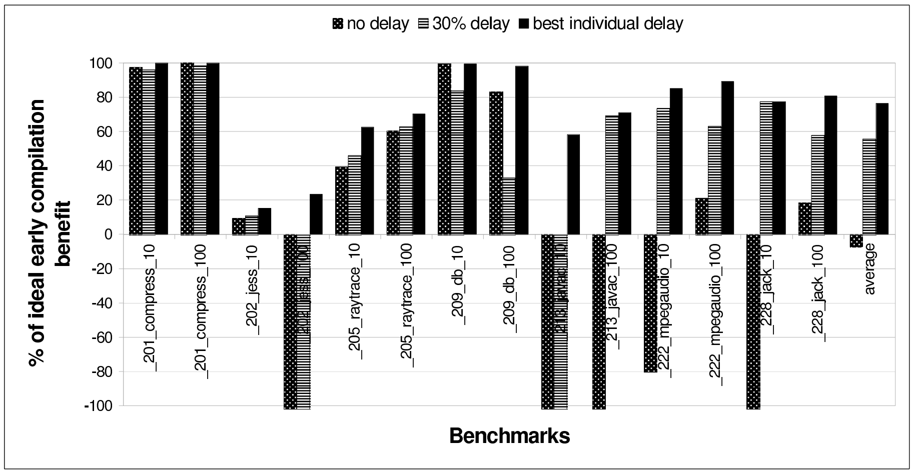


Figure 6.4. Individual benchmark performance results of delaying compilation to improve prediction accuracy, as a percentage of ideal early compilation benefit at selected delay factors

Figure 6.4 shows the percentage of ideal early compilation improvement that is achieved for each benchmark at selected delay factors. Thus, small delay factors seem to do well for most benchmarks. At the same time, elimination of false positives also seems to be very important to achieve close to ideal performance gains.

Chapter 7

Feasibility of Early Determination of Loop Iteration Counts

In the last chapter we showed that the comprehensive knowledge of loop iteration bounds can allow us to achieve close to the ideal performance benefits of early compilation for several benchmarks. However, the success of this technique still hinges on the ability of the static analysis framework and, more importantly, the managed runtime environment to automatically determine the iteration bounds of program loops. In this chapter, we explore the feasibility of determining the loop bounds for only the application classes during execution, and the impact of non-analyzable loops on the early detection of method hotness, and the overall performance improvement.

Figure 7.1 presents an instance of each of the three categories of loops that we currently label as *analyzable*, since the iteration bounds of such loops can be deciphered at runtime before entry into the loop. The iteration bound of loops in category A can be trivially determined statically or dynamically. Even though, the loops in category B may depend on runtime values, their iteration bounds can also be determined dynamically. We modified the *branch* routine in the VM interpreter to confirm that the bounds of loops in categories A and B can indeed be determined prior to the first entry into the loop. Loops belonging to category C are simple loops that iterate over standard library data structures, and will most probably require additional static analysis and/or classfile annotations to be analyzable. We still categorize such loops as analyzable since we believe that capability to prepare such loops for our prediction tasks, such as by adding additional instructions to indicate the loop bound to the VM (shown by the bold comment for Loop C), can be made available to static analysis tools, such as Soot.

```
for(i=0 ; i<25000 ; i++){
    static_hot();
}
```

Category A

```
n = data.length;
for(i=0 ; i<n ; i++){
    dynamic_hot();
}
```

Category B

```
void method(List list){

   // n = list.size();
   while(it.hasNext()){
      el = (Element) it.next();
   }
}
```

Example 1

```
while((line = in.readLine()) != null) {
}        ...
```

Example 2

Category C

Figure 7.1. Iteration bounds of some categories of loops can be accurately predicted early

On the other hand, loops in Figure 7.2 belong to categories that can make it highly improbable or very expensive to a priori determine their bounds. Loops in category D iterate over program-specific data structures, and category E loops are non-linearly dependent on input values. We term such loops as *non-analyzable* in this thesis.

We performed a manual study using the Soot framework to find the percentage

```
do{
    diff_func(myStack.pop()
} while(!myStack.isEmpty())
```

Category D

```
tok = getToken();
i=0;
while(arr[i] != tok){
    ...
}
```

Category E

Figure 7.2. Iteration bounds of some categories of loops are difficult to predict

Benchmark	Analyzable Loops			
	Category			% of
	A	B	C	Total
_201_compress	1	15	2	69.23
_202_jess	1	140	4	84.79
_205_raytrace	9	21	2	74.42
_209_db	2	12	1	71.42
_213_javac	3	108	41	64.13
_222_mpegaudio	23	43	2	88.31
_228_jack	2	29	37	76.40

Table 7.1. Loops that permit determination of its iteration count prior to entry

of analyzable loops in the SPECjvm98 benchmark programs. Table 7.1 lists the results of this study. For each benchmark, columns two, three, and four present the number of loops in categories *A*, *B*, and *C*, while the last column lists the percentage of total analyzable loops. Thus, a large majority of the loops in most benchmarks can be easily targeted by our approach.

Table 7.2 and Figure 7.3 show the results of discarding the non-analyzable loops from the simulation runs, and correspond to the numbers presented earlier in Table 6.1 and Figure 6.3 respectively. More than affecting the detection of the actual hot methods, eliminating the non-analyzable loops has the unexpected side-effect of reducing the number of false positives. The number of false positives for different delay factors is presented in Table 7.2.

Benchmark	False Positives at Delay Factors (% of 10,000)										
	0%	1%	3%	7%	10%	20%	30%	40%	50%	70%	90%
_201_compress_10	2	2	**0**	0	0	0	0	0	0	0	0
_201_compress_100	3	1	**0**	0	0	0	0	0	0	0	0
_202_jess_10	3	3	3	3	1	**0**	0	0	0	0	0
_202_jess_100	25	17	13	**11**	9	6	6	4	4	4	0
_205_raytrace_10	18	5	**4**	4	4	4	2	2	2	0	0
_205_raytrace_100	7	5	**4**	4	4	4	2	2	2	0	0
_209_db_10	**1**	1	1	0	0	0	0	0	0	0	0
_209_db_100	**3**	3	2	0	0	0	0	0	0	0	0
_213_javac_10	55	54	51	36	36	35	30	23	13	**5**	0
_213_javac_100	207	126	105	84	71	15	**0**	0	0	0	0
_222_mpegaudio_10	20	13	7	4	**2**	1	1	1	0	0	0
_222_mpegaudio_100	44	9	**5**	1	1	1	1	1	1	1	0
_228_jack_10	9	7	**0**	0	0	0	0	0	0	0	0
_228_jack_100	14	10	2	**0**	0	0	0	0	0	0	0

Table 7.2. False positives at different delay factors after removing non-analyzable loops

Not surprisingly, discarding non-analyzable loops also delays the early detection of hot methods in some cases. The average performance improvement over the default VM performance is presented in Figure 7.3. The combination of reduced instances of false positives and delayed detection of *actual* hot methods, results in improving the performance over the all-loops configuration of figure 6.3 for the lowest delay factors, but results in some degradation for the remaining factors. The most prominent reduction in the performance gain is witnessed for the *_201_compress* benchmark, in which case the improvement dropped from over 33% to about 19% over the default VM performance. Overall, we measured an average improvement of 66% of ideal offline early compilation benefit considering the *best* delay factor for each benchmark.

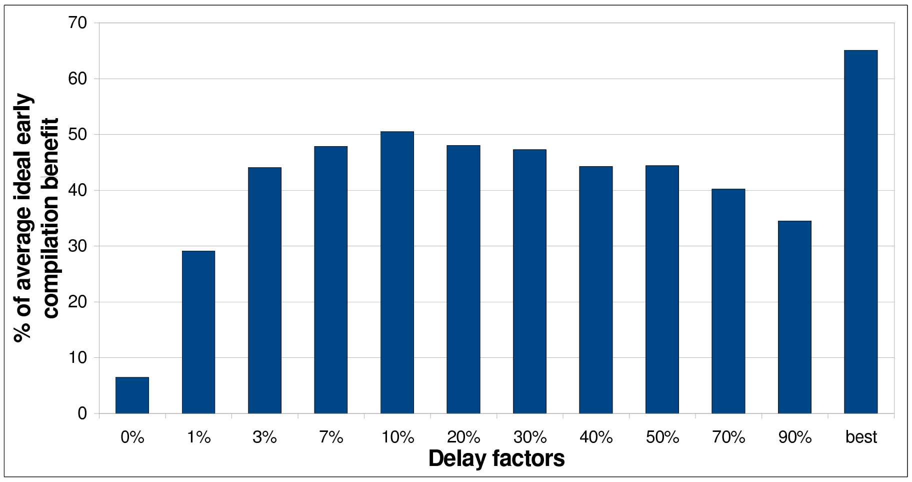


Figure 7.3. Average performance gain over all benchmarks at different delay factors after removing non-analyzable loops

7.1 Implementation Cost

In order to remain completely transparent to the user, the online profiling approach described in this work should be implemented *entirely* at runtime. At the same time, for dynamic compilation to improve performance, it is critical that the profiling and decision mechanisms incur low overhead so as to not subsume the benefits of early compilation. To minimize execution-time overhead while maintaining complete user transparency, we suggest the following implementation strategy for our new profiling mechanism: (1) During classfile loading, a simple analysis pass will scan the input bytecodes to identify and mark loop entry/exit instructions for the VM to generate appropriate trace *events* during interpretation. Thus, this pass will replace the static analysis and classfile annotation phase currently performed by Soot during our trace-file simulation (Step 1 of Section 5.1). If needed, this dynamic analysis pass can be performed in a separate thread (on a free core) to minimize overhead in the primary application thread. (2) The next stage of the profiling mechanism (described in Steps 2-3 of Section 5.1), which generates the trace information, should require extremely low overhead and can be performed inline during the interpretation of the main application. The events generated by this stage will be inserted into a trace queue. (3) The last stage of our profiling mechanism (described in Section 5.2) is the higher overhead decision making component that reads (and removes) events from the trace queue and determines method hotness. This component should, ideally, be implemented in a separate thread and run on a free core to minimize interference with the main application execution.

Thus, by minimizing overhead in the main application thread, the above implementation strategy should be able to hide any additional overhead imposed by our new profiling mechanism on modern machines. Implementation strategies for online profiling that employ a distinct profiling thread have already been successfully attempted in other VMs [2]. At the same time we are also exploring heuristics to further reduce the profiling overhead, such as only generating events for loops with *large* loop bounds to reduce the number of events generated without significantly affecting the profiler's view of future behavior. For example, focusing on loops with an iteration bound greater than 10 reduces the number of dynamic loops entered by almost 80%, but only drops the best average performance in Figure 7.3 by 3.4%.

Chapter 8

Future Work

There are a variety of enhancements that we plan to make in the future.

8.1 Future implementation steps

We are currently implementing our profiling technique along with the best heuristics found during our current analysis in a real VM using the implementation strategy suggested in Section 7.1. Actual implementation in the HotSpot VM will enable us to measure the overhead imposed by our approach, and determine the potential benefit of a purely online early compile implementation. Following are the stages in which we plan to proceed with our research:

1. Modify the interpreter to generate the method invocation and loop entry events, and insert those in a global queue. Measure the performance gain with this overhead of additional instructions corresponding to the numbers presented earlier in Figure 7.3. These results will measure the overhead imposed in the main application thread.

2. We aim to simplify and reduce the events that are necessary to implement

our post analysis algorithm. We can assign the loop identifiers based on the nesting level and approximately determine when the loop has exited. Figure 8.1 shows the example source program and the generated trace file. The nesting level of the loops can be determined statically and the loop ids

```
class  TestProgram {
    // MethodID = 100
    public static void main(String args[]) {
        // LoopID = 1000
        for(int i=0 ; i<10 ; i++) {
            int j=0;
            // LoopID = 2000
            while(j < args.length) {
                ...
            }
        }
        // LoopID = 1001
        for(int i=0 ; i<20 ; i++)  {
            ...
        }
    }
}
```

(a) Java Source Program

```
f100  $1000 $2000 ... $1001
```

(b) Partial trace file with reduced events

Figure 8.1. Example Java program and trace file with the reduced events

can be assigned accordingly. The outer loops are assigned ids starting from 1000. The loops with the nesting level of 2 are assigned ids starting from 2000 and so on. Thus, with the event \$1001 in Figure 8.1(b), it can be inferred that loops \$1000 and \$2000 have exited and thus exit events need not be produced. With the trace file containing the reduced set of events, it would be interesting to check if the simplified events affect the early method hotness detection ability of the post analysis program.

3. Finally, run the profiler thread on a different core and measure the performance by combining the previous steps. This profiler thread will read the

events from the global queue and run the post analysis algorithm to queue the methods for compilation as they are detected hot.

8.2 Prediction of early compilation benefit

We are currently trying to implement a model that may help to statically determine the early compilation benefit possible for a given benchmark.
Let,
N be number of bytecode instructions interpreted during the default run.
n be number of bytecode instructions interpreted during the early compile run.
T be the total execution time of the benchmark for default run.
t be the total execution time of the benchmark for early compile run.

Thus, $N - n$ is the additional number of bytecode instructions that are interpreted by the default run over the early compile run. This difference is directly proportional to the amount of time saved by the early compile technique over the default, which is $T - t$. Now, for any other benchmark, if we have N' and n' as number of bytecode instructions interpreted by default and early compile runs respectively and timesaved' is the amount of time saved due to early compilation, then following should hold true:

$$timesaved' = \frac{(N' - n') * (T - t)}{(N - n)} \tag{8.1}$$

Note that considering the difference, $T - t$, makes the equation independent of the characteristics of a benchmark.

$T = default\_appl\_time + compilation\_time + IOTime$

$t = early\_appl\_time + compilation\_time + IOTime$

The time required to compile the hot methods remain constant for the default (T) and early (t) compile configurations. Also, the time spent by the benchmark doing Input-Output operations is the same for both the configurations. Thus, the difference (T-t) is the amount of application thread time that is saved by the early compile run.

We have attempted to verify if equation 8.1 holds true for the benchmarks in our set. But, we found that the experimentally recorded early compile benefit significantly differs from the calculated value. Therefore, we want to further investigate the factors causing this difference. In Equation 8.1, we make the simplifying assumption that every instruction has the same execution time. But, this assumption may not be true. Early compilation for some benchmark might have a greater benefit if it prevents interpretation of more slower bytecode instructions than some other benchmark. In the future we will measure the number of bytecodes saved from interpretation for each type of bytecode to fine tune the calculation. Secondly, we will also apply the *"leave out one cross-validation"* technique to verify the accuracy of this model. That is, we will average out the ratio of (T-t) and (N-n) over $(N_{bm} - 1)$ benchmarks and use this average value to calculate the time saved for the left out benchmark. Note that since we are using absolute times, we need to ensure that all the benchmarks run on the same machine.

8.3 Miscellaneous future work

We are working to expand our benchmark set to include newer and greater number of programs. In fact, preliminary results on the SPECjvm2008 benchmarks show an ideal performance improvement of over 10% due to early compilation of hot methods. Second, the success of our approach depends on the ability to determine loop iteration bounds early and accurately. Therefore, we plan to evaluate various static and dynamic techniques, along with program transformations to dynamically analyze Category C loops, as well as expand our existing set of analyzable loops, and investigate other approaches of detecting future method hotness behavior. Third, we plan to explore the area of automatically finding the best delay factor to use on a per-benchmark or even per-method basis to achieve the most performance benefit. We plan to explore using different confidence measures for different categories of loops (A, B or C), or use other machine learning techniques to predict the best delay factors in individual cases. Finally, we believe that the concept of employing a managed runtime environment to *see* future program execution behavior dynamically is the most significant contribution of this work. Consequently, we plan to apply this technique to other areas, including garbage collection, security, and other aspects of performance improvement.

Chapter 9

Conclusion

In conclusion, we can say that our exploration into this novel approach of online profiling shows mixed, but promising, results for selective compilation. We showed that the current *reactive* mechanisms to online profiling suffer from major drawbacks, including incorrect hot method speculation and delays in making the associated compilation decisions. These drawbacks result in considerable performance losses during program startup on our benchmark programs. Our novel profiling strategy, based on the hypothesis that early knowledge of loop iteration bound information can allow an online profiler to determine future program behavior, producing early and accurate compilation decisions, allows early hotness detection for most benchmarks, but with several false positives in many cases. Interestingly, simple heuristics are able to eliminate almost all false positives for most benchmarks without much degradation in performance. Although further studies show that our new online profiling approach is feasible for current benchmarks, the VM may need to add capabilities of analyzing more loops for maximum benefit. We believe that our suggested plan for online implementation of our new profiling strategy is practical and cost-effective for current VMs and architectures.

Additionally, we also believe that the core concept of employing the virtual machine to understand and exploit future program behavior shows promise, and can also be applied to several other areas of computer science.

References

[1] A. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 267–276, 2004.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, 2000.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, February 2005.

[4] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. *SIGPLAN Not.*, 37(11):111–129, 2002.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, 2000.

[6] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 13–20, 2000.

[7] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. D. Bosschere. Using hpm-sampling to drive dynamic compilation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 553–568, 2007.

[8] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21:1301–1321, 1991.

[9] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 13–26, New York, NY, USA, 2000. ACM.

[10] M. Corporation. *Microsoft C# Language Specifications.* Microsoft Press, first edition, April 25 2001.

[11] D. Detlefs and O. Agesen. The case for multiple compilers. In *OOPSLA'99 Workshop on Peformance Portability, and Simplicity in Virtual Machine Design*, pages 180–194, November 1999.

[12] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[13] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. In *ASPLOS-X: Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*, pages 202–211, New York, NY, USA, 2000. ACM.

[14] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220, 2003.

[15] A. Gal, M. Bebenita, and M. Franz. One method at a time is quite a waste of time. In *Proceedings of the Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2007.

[16] M. X. Goemans. Advanced algorithms. Technical Report MIT/LCS/RSS-27, 1994.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (3rd Edition)*. Prentice Hall, third edition, June 14 2005.

[18] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Javatm just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the conference on Virtual Machine Research And Technology Symposium*, pages 12–12, 2004.

[19] G. J. Hansen. *Adaptive systems for the dynamic run-time optimization of programs.* PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1974.

[20] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 253–264, Washington, DC, USA, 2003. IEEE Computer Society.

[21] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43, 1992.

[22] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[23] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.

[24] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. W., R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, 1993.

[25] R. M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *Proceedings of the IFIP World Computer Congress on Algorithms, Software, Architecture - Information Processing, Vol 1*, pages 416–429, 1992.

[26] D. E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[27] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 156–167, 2001.

[28] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, December 2000.

[29] P. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 94–104, 2007.

[30] M. Mock, C. Chambers, and S. J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 291–302, 2000.

[31] M. Paleczny, C. Vick, and C. Click. The java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[32] K. Pettis and R. C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, 1990.

[33] P. Pominville, F. Qian, R. Vallée-Rai, L. J. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 334–354, London, UK, 2001. Springer-Verlag.

[34] SPEC98. Specjvm98 benchmarks. http://www.spec.org/jvm98/.

[35] K. D. Team. Kaffe java virtual machine. http://www.kaffe.org/, September 19 2007.

[36] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13, 1999.