Novel Online Profiling for Virtual Machines

Manjiri A. Namjoshi Prasad A. Kulkarni
Department of Electrical Engineering and Computer Science, University of Kansas
{manjiri,prasadk}@ku.edu

Abstract

Application *profiling* is a popular technique to improve program performance based on its behavior. *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 paper 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.

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

General Terms Languages, Performance

Keywords Virtual machines, Online profiling, Java

1. Introduction

Application profiling gathers information regarding program characteristics during execution, and is a popular technique to understand and reason about the dynamic behavior of a program [Graham et al. 1982, Chang et al. 1991]. Profile information is used to identify important execution patterns that are further employed to dispatch important methods for dynamic compilation, as well as to tailor feedback-directed optimizations to improve program performance. Profile information is collected using mostly one, or a combination of two strategies: (1) additional prior runs of the same program (*offline* strategy), or (2) dynamically, during the current program run (*online* strategy). Profile-based compilation and optimization, when feasible and successful, can result in significant performance benefits.

Offline profiling captures program behavior from *previous* program runs to guide optimization decisions for future runs [Hwu et al. 1993, Chang et al. 1991, Mock et al. 2000, Pettis and Hansen 1990]. Although, the resulting performance improvements are often significant, offline profiling 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(s).

Managed language runtimes (also called Virtual Machines (VM)), such as those for Java and C#, offer a unique opportunity to overcome the drawbacks of offline profiling by providing an environment to adaptively perform application profiling online [Arnold et al. 2002, Hazelwood and Grove 2003, Arnold et al. 2000]. Indeed, online profile-directed compilation is considered critical to the *startup* as well as the overall performance delivered by current VMs. However, current online profiling mechanisms only monitor the *past* application behavior during the current run to predict future behavior of the same run. Moreover, gathering this profile information typically requires current approaches to monitor program information over a substantial amount of time. Accordingly, in spite of their success, in this paper we show that present implementations of online profiling still need to address several important challenges:

1. revise their *reactive* program monitoring strategy that causes associated adaptive transformations to base their decisions on partial and stale profile information, and
2. reduce the time spent waiting for sufficient profile information to be collected that *delays* critical compilation and optimization decisions at program startup.

Arguably, the most prominent application of online profiling in virtual machines is during *selective compilation* to detect the subset of *hot* methods to selectively compile at program startup [Hölzle and Ungar 1996, Paleczny et al. 2001, Krintz et al. 2000, Arnold et al. 2005]. A method is designated as hot if the application spends a significant proportion of its runtime in the corresponding section of code. Selective compilation is an extremely important and universally adopted technique to limit the overhead of adaptive compilation, while deriving the most performance benefit at runtime. The drawbacks of current online profiling implementations result in several important consequences for selective compilation. First, adaptive profile-driven compiler transformations can, counter-intuitively, lead to performance degradation (by spending time in optimizing inconsequential sections of code) if the profile does not correctly speculate future program behavior. At the same

time, applications typically notice poor startup performance due to execution in unoptimized code or via interpretation, while the adaptive optimization system is waiting for profile information to be collected and associated transformations to be performed. In this paper we demonstrate that the above consequences seriously impact application performance for several benchmarks at startup.

To address these drawbacks we propose a new online profiling mechanism for VMs that attempts to read the values of *loop iteration bounds* before every loop entry to predict the future hotness status of all methods invoked in that loop, and prior to their first invocation. The compilation and optimization of hot methods can now be triggered early and be guided by the knowledge of their future behavior. Such a technique can lead to reduced misspeculations along with the benefits of early compilation. The goal of our present work is to explore the potential, feasibility, and benefits of this new profiling mechanism during selective compilation. Additionally, we also suggest a low-cost and completely online mechanism for its implementation in process VMs.

Thus, the main contributions of this paper are:

1. explain the drawbacks of current implementations of online profiling in virtual machines,
2. demonstrate their performance impact on program startup during selective compilation,
3. propose a new online profiling strategy that can gather and employ knowledge of future application behavior to guide dynamic compilation and optimization decisions,
4. evaluate the performance potential of complete as well as practical knowledge of loop iteration bounds to predict future method hotness, and
5. study issues regarding the feasibility and runtime cost of our new profiling strategy for selective compilation.

The rest of this paper is organized as follows. We describe work related to the areas of online profiling and selective compilation in Section 2. We outline our experimental framework in Section 3. We demonstrate the drawbacks of current implementations of online profiling in Section 4. We present our simulation-based experimental framework, and evaluate the potential and feasibility of our novel online profiling strategy for selective compilation in Section 5. We outline our plan for the future maturation of this online profiling framework in Section 6, and finally draw our conclusions in Section 7.

2. Background and Related Work

The requirements of code portability and dynamic verifiability prevent the application of powerful static optimizations for programs written in managed languages. Therefore, such programs rely extensively on accurate online profiles and powerful runtime optimizations for their performance needs. In this section we provide background information and related work in the areas of online profiling and selective compilation.

In most situations, executing native optimized code is several orders of magnitude faster than interpretation. However, by performing their compilations at run-time, managed environments can potentially increase the *total* execution time of native execution over interpretation if the compilations are performed injudiciously. Consequently, most dynamic environments employ several mechanisms to reduce the overhead of dynamic compilation at runtime. Selective compilation is a technique devised to reduce the compilation overhead by exploiting the well-known fact that most programs spend most of their time in a small part of the code [Knuth 1971, Bruening and Duesterwald 2000, Arnold et al. 2005]. This observation allows selective compilation to focus resources on sections of code that are frequently-executed or *hot* [Hansen 1974, Grcevski et al. 2004, Team 2007, Detlefs and Agesen 1999]. Thus, virtual machines employing selective compilation only optimize the hot sections of code, and use interpretation [Grcevski et al. 2004, Kotzmann et al. 2008] or naive compilation [Cierniak et al. 2000, Arnold et al. 2000] to execute the rest of the program. *Profiling* is used during selective compilation to identify the candidate hot code regions for compilation and promotion to higher levels of optimization.

Current approaches to dynamically select hot methods for compilation use profiling techniques that are typically based on *counters* [Hansen 1974, Hölzle and Ungar 1996, Kotzmann et al. 2008] and *sampling* [Arnold et al. 2000, Grcevski et al. 2004]. The approach based on counters increments a counter on each method invocation and, optionally, on each loop iteration. The other mechanism uses sampling to periodically interrupt the system and update a counter for the method(s) on top of the stack. If the counter reaches a pre-determined threshold, then the method is queued for compilation. Researchers have also explored the use of hardware performance monitors to reduce the overhead of collecting profile information [Adl-Tabatabai et al. 2004, Buytaert et al. 2007]. However, it is generally hard to associate the very low-level information obtained via hardware monitors with the higher-level program elements that actually influence the counters.

All these popular profiling mechanisms are typically reactive and speculate their compilation and optimization decisions based on profiles of past application behavior, which may be inaccurate [Duesterwald et al. 2003]. Accordingly, compilation in most current runtime systems may even result in a performance loss if their speculation goes wrong. Performance degradations due to incorrect speculations are commonly seen, especially for short-running applications and for program startup performance. Moreover, a virtual machine employing current profiling approaches to detect hot methods needs to postpone its compilation decisions until sufficient profile information has been collected to determine method hotness. 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. In contrast, our proposed counter-based profiling mechanism is an attempt to make optimization decisions early and based on guarantees regarding the future application behavior.

Static analysis of the source code has been employed in earlier works to send hints to the JIT compiler to reduce its compilation overhead, and enable powerful and time-consuming optimizations at runtime [Hummel et al. 1997, Azevedo et al. 1999, Krintz and Calder 2001, Pominville et al. 2001]. Krintz combined offline and online profiling to overcome some of the drawbacks in both profiling approaches [Krintz 2003]. All these techniques are dependent on some combination of prior offline profiling runs, classfile annotation, and annotation compression. Instead, we propose a completely online profiling approach in this paper. Other researchers have also demonstrated that delaying important compilations can be harmful to application startup performance [Kulkarni et al. 2007, Harris 1998]. However, their work only focused on reducing the delay between the detection and compilation of hot methods, or the time spent waiting in the *compilation queue*. Instead, our work presented in this paper attempts to enable earlier and more accurate detection of hot sections of code.

3. Experimental Framework

The experiments in this paper are conducted using the SPECjvm98 suite of benchmarks [SPEC98]. Table 1 lists the name and relevant features for each of our eight SPECjvm98 benchmarks. All benchmarks come with a small(10) and a large(100) input size, providing us with 16 distinct benchmark-input pairs. This size is indicated along with the benchmark name in the first column of Table 1. The other columns list the total number of methods executed, number

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 1. Benchmarks used in our experiments

of total (application+library) and just application methods that are detected hot, and the number of static application loops for each benchmark-input pair.

We employ the OpenJDK Java Virtual Machine (JVM) and the associated high-performance optimizing *HotSpot* JIT compiler (build 1.7.0-ea-b24) [Kotzmann et al. 2008] to conduct our experiments for this paper. By default, the JVM is configured to operate in two modes. The JVM initially starts in the interpretation mode to minimize the program response time at application startup. The HotSpot VM uses a counter-based profiling mechanism, and uses the sum of a method's *invocation* and loop *backedge* counters to detect and promote hot methods for compilation. Methods are determined to be hot if the sum of the method invocation and loop backedge counts exceeds a fixed threshold. The tasks of detecting hot methods and dispatching them for compilation are performed at every method call.[1]

Choosing the right hotness threshold is very critical to the startup performance of the JVM. A high threshold may result in the VM missing opportunities to detect and compile hot methods, while a low threshold may send even the unimportant methods for compilation, thereby unnecessarily increasing the compilation overhead. Total application performance is negatively impacted in both cases. 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 method for detecting hotness thresholds to use in production JVMs. The results of our experiment for a subset of the tested thresholds (.15X, 0.5X, X, 1.5X, 2.5X) are presented in Figure 1 (Here, the best threshold, and the one we selected for our experiments, is indicated by **X**). In this figure, each benchmark's performance at the threshold of X is used as the base to compare its performance at the remaining thresholds. We used the results of this experiment to determine the threshold achieving the best average performance for our set of benchmarks, and selected that as our baseline compilation threshold.

[1] HotSpot also uses the expensive and complicated *on stack replacement (OSR)* strategy to promote methods with long-running loops to compilation at much larger loop backedge counts. Due to the complexity of the OSR process [Hölzle et al. 1992, Gal et al. 2007], OSR compilation, in most cases, is only supported in commercial VMs and some other extensive research projects such as Jikes RVM [Arnold et al. 2000]. 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. For this paper, we disable the OSR compiles in HotSpot to keep our study simple and to allow more straight-forward analysis of our results.

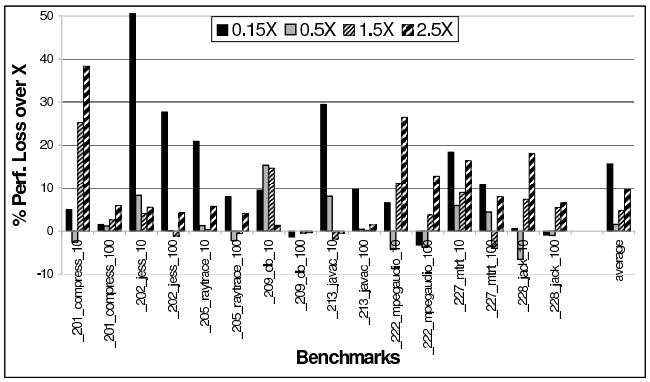

Figure 1. Comparison of benchmark performances at different hotness thresholds with the threshold of X

Some of the studies presented in this paper require a static analysis pass over the java classfile, or require annotating methods in the classfile to convey static or profile information to the JVM. We conduct the static analysis and annotation of the application classfiles by extending the Java bytecode analysis and annotation framework, Soot [Vallée-Rai et al. 1999]. All our experiments are performed on a single core Intel(R) Xeon(TM) 2.4GHz processor using Fedora Linux version 7 as the operating system. Finally, to account for inherent timing variations during the benchmark runs, all the performance results in this paper report the median over 10 runs for each benchmark-configuration pair.

4. Drawbacks of Current Approaches to Online Profiling and Compilation

Most performance-aware JVMs employ selective compilation to dynamically compile hot methods to improve application performance at minimal compilation overhead. In most VMs, the determination of hot methods is performed entirely at run-time in order to take advantage of the execution profile of the program. In this section we evaluate the performance impact due to the drawbacks of current reactive online profiling strategy used to select methods for compilation. We explore drawbacks from the following issues:

1. the past method hotness behavior may not be representative of future method execution behavior, and
2. the time spent in profiling the application to detect hot methods delays their compilation, and thus reduces program start-up performance.

Selective compilation currently employs profiling to find the subset of methods that were hot in the current run so far, and sends them for compilation based on a speculation that they will continue to remain hot in the future. Thus, the methods that are compiled are those that were hot in the past, with no guaranty of their future hotness. It is currently believed that this speculation turns out to be correct in most of the cases. However, we find this to not be the case.

In order to find the number of incorrectly speculated methods during the execution of our benchmark programs we turn to the theory behind the technique of choosing a compile threshold for selective compilation. The theory entails that in the absence of informa-

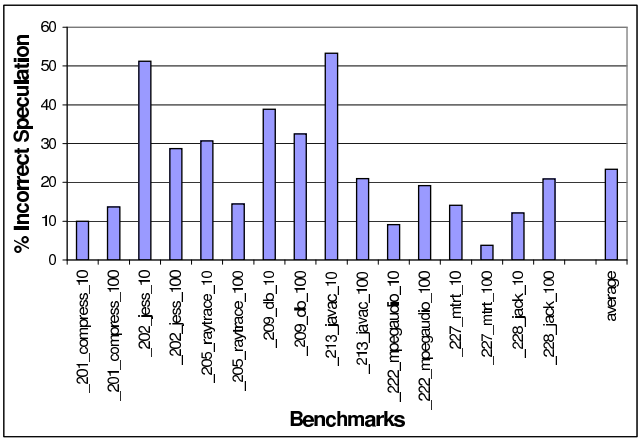

Figure 2. Performance improvement after removing incorrectly speculated methods from the list of hot methods compiled at runtime

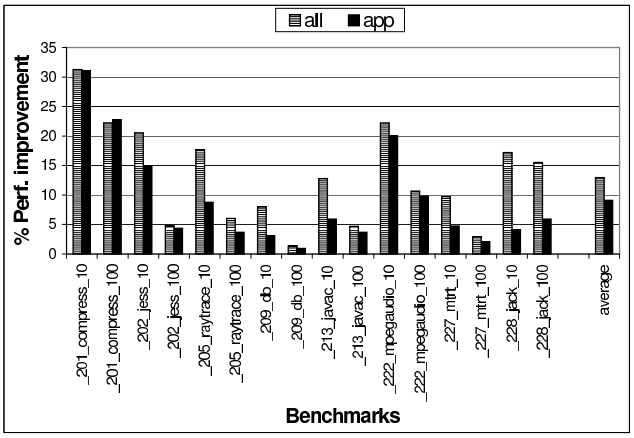

Figure 3. *Ideal* benefits of early compilation of hot methods

tion regarding the future execution behavior of a method, a selection policy that minimizes the worst-case damage of online compilation (in case the speculation goes wrong) will achieve the best performance (sometimes called the *ski-renting principle*) [Goemans 1994, Karp 1992]. According to this principle, to minimize the worst-case damage, a method should only be compiled after it has been interpreted a sufficient number of times so as to already offset the compilation overhead. In the absence of a guaranty regarding the number of future method executions, this approach results in a *worst-case* overhead of twice the amount of time spent in compilation. The worst-case is reached if the compiled method is never executed after being compiled. Any other selection strategy will result in a greater worst-case damage. Thus, mathematically, if the time required to execute the compiled method is P, the time required to interpret the same method is Q, and the method is executed *n* times, then, the only requirement for compiling a method should be,

$$nP + compilation\ overhead < nQ \quad (1)$$

However, since current approaches are unable to a priori determine *n*, VMs use the ski-renting principle to compile a method after it has already been interpreted *m* times, and the following equation is estimated to be true:

$$mP + compilation\ overhead = mQ \quad (2)$$

It is expected that the empirical procedure typically followed to determine the best average threshold in Figure 1 over a range of benchmarks will approximate this theory to yield the best selection policy for each compiler in a VM.

We employ the ski-renting principle and Equation 2 to reason that a compilation decision is an instance of incorrect speculation if a method that is interpreted I number of times before compilation executes for C number of times after compilation, and $C < I$. Thus, in our case, if a method that is detected hot and compiled at a hotness threshold of X has an overall (invocation + backedge) count of less than 2X, then we consider the detection as a case of incorrect speculation. These results, presented in Figure 2, show that using past hotness behavior to guide future compilation decisions can turn out to be incorrect in a significant percentage of cases. More than 23% of methods, on average, are incorrectly sent for compilation, and the incorrect speculation is as high as 53% for some benchmarks. Incorrect speculation of hot methods implies that for these methods the performance gain due to execution in optimized native code may not be able to offset the compilation overhead. Indeed, preventing the wrongly speculated methods from being compiled results in a 3.79% increase in average performance.

The other drawback of current profiling approaches is the delay they introduce in making the compilation decisions due to the time spent in collecting the necessary profiling information. Currently, the VM is required to interpret each method and collect profile information for at least the first *m* method invocations (from Equation 2). Depending on the compiler configuration this *m* can be quite large. Not surprisingly, interpreting hot methods for so long is very detrimental to the startup performance of the application.

We devised experiments to quantify the performance loss caused by this profiling delay in compiling the hot methods. Our strategy uses offline profiling to determine the set of hot methods, and then annotates the classfiles using Soot to indicate them to the VM. Our modified VM recognizes and compiles the annotated methods at the indicated counts. Figure 3 compares the performance results when compiling only the hot application methods (*app*) [2] as well as *all* (application+library) hot methods early at a hotness count of 1, to compiling them as normal at their default hotness threshold (X). Thus, we found early compilation of *all* the hot methods to result in a performance benefit of over 14.5%, on average, over the default technique. Note that early compilation using this offline strategy also eliminates incorrect method speculations. Note also that since we are only impacting the startup performance, the benefits for smaller (input size of 10) benchmarks (17.7%) are much better than those for longer-running (input size of 100) benchmarks (11.4%).

Thus, our results in this section clearly indicate that the current approach of online profiling for selective compilation results in sub-optimal performance. One approach to overcome the issues identified here is to use *offline* profiling to pre-determine the set of hot methods, annotate the classfiles statically, and modify the VM to recognize the annotations and compile the indicated methods early [Krintz and Calder 2001] (as done in the experimental strategy for Figure 3). However, this approach suffers from the issues with offline profiling mentioned earlier, is not transparent to the developer/user, and is consequently not popular, as witnessed by the lack of offline profiling support in most mainstream VMs. In the following sections, we propose and evaluate the potential and feasibility of a new *online* profiling approach that is based on monitoring the values of key program variables in the VM to determine the future method hotness behavior. Methods can then be sent to compilation early and more accurately to achieve the performance

[2] We use this number as the baseline for some of our later experiments.

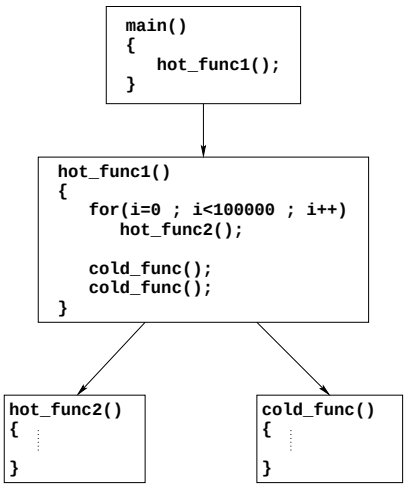

Figure 4. Simple partial program flowgraph

gains close to the ideal offline benefits demonstrated in this section, but without incurring any of the associated offline profiling drawbacks.

5. Loop Iteration Counts for Early Prediction of Method Hotness

The experiments presented in the last section show that correct detection of future method hotness and early compilation of hot methods can significantly benefit the startup performance of managed language applications. Our proposed technique to achieve these benefits at runtime seeks to read the loop iteration bounds before the loop is entered to determine the future execution frequency of all methods early.

Figure 4 shows a hypothetical program call-graph consisting of four functions to demonstrate our intuition regarding how information of all loop iteration bounds can accurately determine the method hotness status early. The method `main` is only called once and contains no loops. Consequently, execution will not stay in this region of the program for very long, and hence `main` is not a hot method. Method `hot_func1` is hot since it contains an important loop in its function body. Method `hot_func2` is hot because it is invoked from a hot loop. In contrast, although method `cold_func` is called from a hot method it is not itself hot since it is not called from any loop, and does not contain a loop in its body. Thus, exhaustive loop bound information may be sufficient to predict the future method hotness. However, even with complete loop bound information, detection accuracy might be affected in some cases due to, for example, dependence on conditional program control-flow or values of program input.

Our proposed online profiling technique is based on the hypothesis that determination of loop iteration bounds before entering the loop is feasible for most program loops, and allows early detection of hot methods. In this section, we conduct a systematic evaluation of this hypothesis. To limit the complexity of this evaluation, we restrict the scope of this study to only the hot methods and loops present in the application classfiles (ignore hot library methods) and compare our results with the ideal offline *app* performance results in Figure 3. Our evaluation consists of the following steps:

1. In section 5.1, we employ an *offline* trace-based simulation study to determine the potential of *exhaustive*, accurate, and early loop iteration bound information to detect future method hotness.
2. In section 5.2, we analyze why even the most accurate loop bound information may be insufficient in some cases to achieve early compilation benefits, and evaluate a new heuristic to address the identified issues.
3. In section 5.3, we study the feasibility of determining loop iteration bounds early at runtime.
4. Finally, in section 5.4, we describe an implementation approach to perform all the profiling steps at runtime, while minimizing the overhead to the main application thread, to result in an entirely *online* profiling strategy.

5.1 Trace-Based Simulation Study

In this section we describe the results of our offline study to test the hypothesis that early knowledge of *all* loop iteration bounds is sufficient to detect hot methods early. This study is conducted in two stages. The first stage uses a modified VM to generate a trace file that records each loop iteration and method invocation, and makes the loop iteration bound available before each corresponding loop entry. This trace file is analyzed in the next stage to predict the hot methods early. These stages are described in the following two sub-sections.

5.1.1 Simulation Setup

Offline generation of the trace file consists of the following steps:

(1) We employ the Soot bytecode analysis and annotation framework [Vallée-Rai et al. 1999] to identify the loops in every application method, and annotate their corresponding loop entry and exit instructions using *attributes* in the Java classfiles. For example, for the Java source program shown in Figure 5(a), Soot analyzes the three methods, *main*, *methodA*, and *methodB*, and inserts annotations to the Java classfile to indicate the offsets for each *loop entry*, and *loop exit* instruction to the VM. [3]

(2) We then use a modified version of the HotSpot Java virtual machine to recognize our new classfile annotations. On executing the annotated classfiles, our modified JVM prints out *event* markers to identify method entry, loop entry, all later iterations of a loop, loop exit, and the number of iterations for each loop. Figure 5 (b) illustrates the trace file that would be produced after the execution of the annotated bytecode program produced by Soot in the earlier step. We will discuss the individual events in more detail in the next section.

(3) 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 make it available to our offline analysis program on the first iteration of each loop. The trace file from Figure 5 (b) after post-processing appears as shown in Figure 5 (c). The shifted loop iteration bound is indicated in a bold font. Thus, this stage simulates the generation of *events* that could be produced by an appropriately configured VM interpreter at runtime.

5.1.2 Analyzing Trace Events for Early Method Hotness Detection

The second stage in our simulation study is an *analyzer* that interprets each trace event, updates relevant data structures, and outputs the earliest invocation count at which a method is detected to be hot. The analyzer maintains two data structures to facilitate the simulation process:

loop_stack: A single structure to hold information regarding the loop identifier, iteration bound, and the current iteration count for every active loop during program execution. 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 to record the dy-

[3] The classfile generated by Soot is not shown here to conserve space.

```
class TestProgram {
    // MethodID = 1
    public static void main(String args[]) {
        // LoopID = 0
        for(int i=0 ; i<2 ; i++) {
            methodA(10);
        }
    }

    // MethodID = 3
    static void methodB() {
        System.out.print(" ");
    }
```

```
    // MethodID = 2
    static void methodA(int len) {
        int i;
        // LoopID = 1
        while(i < len) {
            if(i%2 != 0)
                methodB();
            i++;
        }
    }
```

(a) Java Source Program

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

(b) 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
```

(c) Post−Processed Trace File

Figure 5. Process of generation of the trace file containing comprehensive information of all loop iteration bounds and method invocations for each benchmark

namic 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.

The trace file generated in Stage 1 contains indicators for four unique events. The example trace file from Figure 5(c) is reproduced in Figure 6(a), and shows each unique event. On occurrence of each of these four events, the analyzer takes the following steps:

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

Loop Backedge: A '`,`' in the trace file indicates the occurrence of a backedge to start the next iteration of the innermost loop (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. The method's invocation count in all relevant loop contexts in the method_info structure is incremented. For the first update of a method's information structure in every new loop context, the simulator estimates the method's total count using the formula:

$$predict\_cnt = (inv\_cnt + back\_cnt) * \frac{bound}{cur\_iter} \qquad (3)$$

where,
inv_cnt is the current method invocation count,
back_cnt is the current method backedge count,
bound is the loop iteration bound, and
cur_iter is the iter. count for the current loop context.

Equation 3 consists of two components. First, $(inv\_cnt + back\_cnt)$ provides the historical information of the total (invocation + backedge) count registered for the method in the first *cur_iter* iterations of the current loop. Next, multiplying this factor by the loop *bound* allows calculation of the estimated future count for the same method. Thus, this equation attempts to answer the following question: If a method has recorded ($inv\_cnt + back\_cnt$) number of counts in *cur_iter* iterations of the current loop, then how many counts (past + future) is the method expected to register in *bound* number of loop iterations. If $predict\_cnt > hotness\ threshold$, then this method will be sent for compilation.

Figures 6(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 snapshots marked in Figure 6(a). Assume for this example that the ***hotness threshold* is 10**. These events are described below:

Event 1, Figure 6(b): The analyzer pushes a new record, consisting of the `loop_id`, the loop iteration `bound`, and the current loop count (`cur_iter`), on the loop stack.

Event 2, Figure 6(c): The fields `inv_cnt` and `back_cnt` in the method_info array, correspond to the number of invocations and backedges already known for each method, and 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 analyzer then employs Equation 3 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 predicted 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.

Event 3, Figure 6(d): 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 3 is again used to calculate `f2`'s predicted count, given by: $(1+10)*2/1 = 22$. Since `f2`'s predicted 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. The invocation

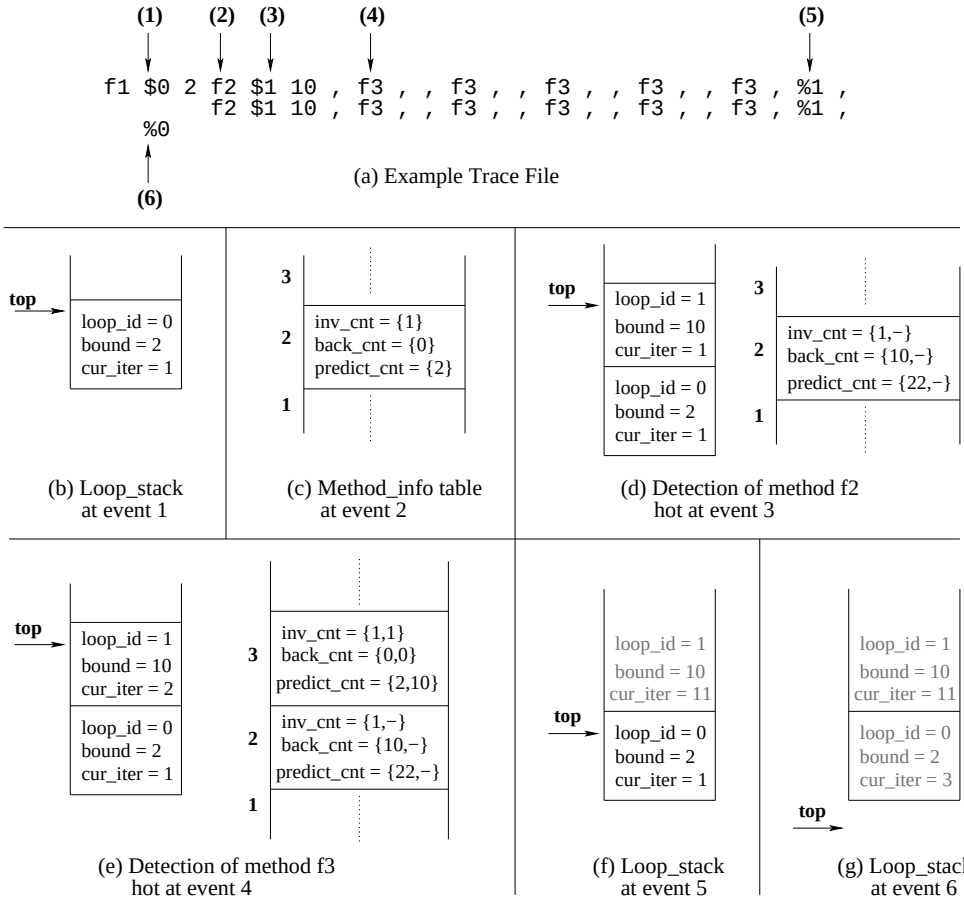

Figure 6. Demonstration of the simulation algorithm on an example trace file (for hotness threshold of 10)

count at which `f2` is first predicted to be hot is output as *1* by the analyzer.

Event 4, Figure 6(e): At this point, the analyzer has already seen the first ',' symbol in the trace file, and accordingly updated the `cur_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 `predict_cnt` 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 predicted 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*.

Event 5, Figure 6(f): The loop record from loop_id=1 is popped off the loop_stack. Note that the *cur_iter* is equal to one more than the *bound* for loop_id=1 at this stage.

Event 6, Figure 6(g): The loop record for loop_id=0 is popped off the loop_stack.

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 the framework used to achieve early compilation in Section 4, but instead of compiling the hot methods on their first invocation, all methods detected to be hot by the analyzer are sent to compile at the indicated counts. Our present simulation framework only supports single-threaded programs, and so we had to leave out the multi-threaded benchmark *_227_mtrt* from the results in this and future sections. However, please note that this is not a limitation for the profiling approach, but only of the current simulation framework.

The analysis results are presented in Table 2 and Figures 7 and 8. The first column in Table 2 lists the benchmark name. The next column, labeled *Actual* shows the actual number of hot methods compiled by the default VM for each benchmark and for both the 10/100 input sizes. Our aim is to predict the hotness characteristic of only these methods early. However, the number of *predicted* hot methods (shown in column three of Table 2) may be greater than their actual number due to *false positives* detected during our trace-file analysis (shown in the column labeled *False Positives* in Table 2). The predicted hot methods is the sum of the actual hot methods and false positives. False positives are *cold* methods that are wrongly predicted hot by our algorithm, and indicate the inability of loop iteration bounds to provide accurate results by themselves. Our current implementation calculates the sum of the (past + future) method counts in Equation 3 to determine hot methods. Therefore, our current implementation does not miss true hot methods. If the *actual* hot methods are not predicted to be hot early, then they will be sent for compilation at the method threshold count of X (although this happens very rarely in practice).

Even for the correct detections, methods may be predicted hot at total counts greater than 1. This unintentional delay seems to be most commonly caused due to inadequate future information available from loops with small iteration bounds. For example, in Figure 9 and for a compile threshold of 10,000, information available

Benchmark	Hot Methods		False
	Actual	Predicted	Positives
_201_compress	17/18	19/22	2/4
_202_jess	22/47	25/51	3/4
_205_raytrace	49/71	67/78	18/7
_209_db	11/9	12/14	1/5
_213_javac	42/309	100/527	58/218
_222_mpegaudio	50/77	159/169	109/92
_228_jack	20/69	63/120	43/51

Table 2. Results of trace file simulation to predict hot methods for both input sizes 10/100

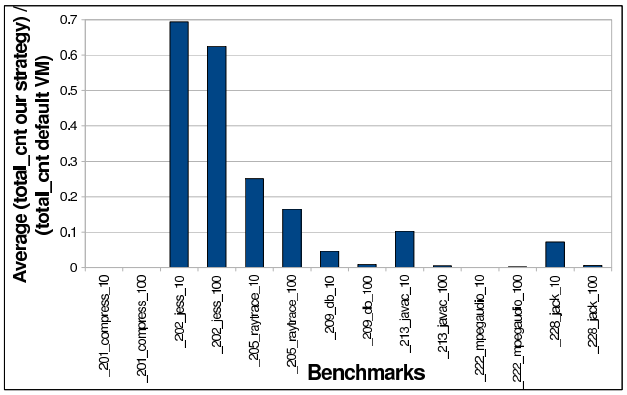

Figure 7. Lower the average ratio, the earlier a method was compiled, on average, compared to the default VM

in `Loop1` regarding the future invocation of method `funcA` is insufficient to predict `funcA` hot ($(1+0)*8000/1 = 8,000$ by Equation 3 is less than 10,000). However, `funcA` will be detected hot in `Loop2`, although, after it has already been invoked for 80 times.

Figure 7 ignores the false positives and plots the average ratio of the total counts at which the *actual* hot methods are first predicted to be hot with our technique as compared to the hotness counts of the default VM profiling strategy. Thus, small ratios for most of the benchmarks indicate that knowledge of loop iteration bounds does allow hot methods in most benchmarks to be detected much earlier than with the default strategy. The presence of mostly recursive methods coupled with very few big loops explains the particularly poor numbers for the benchmark *_202_jess*.

Figure 8 plots the percent fraction of the *ideal* early compilation benefit (from the *app* field in Figure 3) that is obtained for each benchmark when using the analysis results to compile the detected methods early. The results show that the large number of false positives actually produce a significant performance degradation for many of our benchmarks (-0.9%, on average). The inability of our analysis framework to detect hot methods early also results in significantly reducing the gains compared to ideal early compilation, particularly for *_202_jess* and *_205_raytrace*. 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.

5.2 Delaying Early Prediction to Improve Its Quality

Thus, 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 section, we analyze the most common causes for the large number of false positives (cold methods predicted as hot) encountered during our analysis in the last section, and present and evaluate a technique to improve the quality of our predictions.

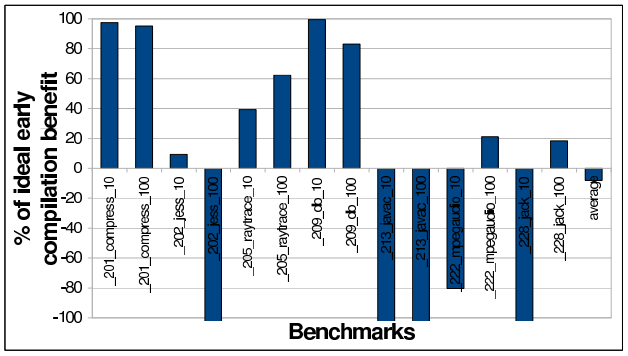

Figure 8. Benchmark performance when compiling methods as predicted by our analysis algorithm compared to the ideal offline performance from the *app* field in Figure 3

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

Figure 9. Small loop bounds delay the prediction of hot methods (for hotness threshold of 10,000)

```
$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 10. Impact of runtime variables on method hotness prediction **(for hotness threshold 10,000)**

The examples (from real benchmarks) presented in Figure 10 indicate the main causes of false predictions. Both the examples in Figure 10 show trace file fragments that are derived from actual benchmark programs. The first fragment explains the effect of variable inner 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

Benchmark	False Positives at Delay Factors (% of baseline threshold)										
	0%	1%	3%	7%	10%	20%	30%	40%	50%	70%	90%
_201_compress	2/4	2/2	**0**/**1**	0/1	0/1	0/1	0/1	0/1	0/1	0/0	0/0
_202_jess	3/4	3/4	3/2	3/2	3/2	3/2	3/2	3/1	3/1	3/0	**0**/**0**
_205_raytrace	18/7	5/5	**4**/**4**	4/4	4/4	4/4	2/2	2/2	2/2	0/0	0/0
_209_db	**1**/5	1/4	1/**2**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
_213_javac	58/218	57/132	53/105	38/84	38/71	37/16	31/**1**	24/1	15/1	6/0	**0**/0
_222_mpegaudio	109/92	14/**4**	8/2	**5**/0	4/0	1/0	1/0	1/0	0/0	0/0	0/0
_228_jack	43/51	41/43	36/**14**	25/0	**10**/0	2/0	1/0	1/0	0/0	0/0	0/0

Table 3. False positives at different delay factors (for both input sizes 10/100)

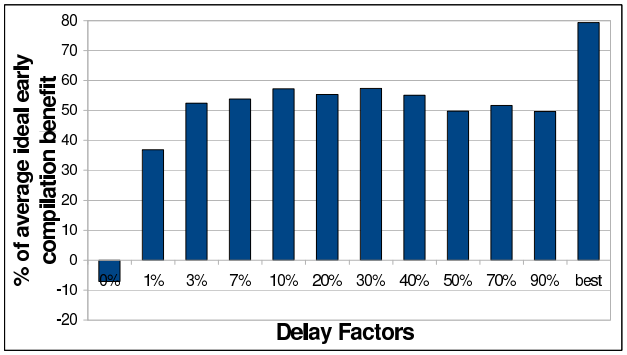

Figure 11. *Average* performance results of delaying compilation to improve prediction accuracy, as a percentage of ideal early compilation benefit at selected delay factors

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 10 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 positive, 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.

To reduce the number of false positives, we propose a 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. The ideal *delay factor* varies for different methods. For example, although a delay factor of 1% (of default hotness threshold) is able to eliminate detection of the false positive for the example in Figure 10, a higher delay factor will be necessary to eliminate the false positive in Figure 10(a). Table 3 shows the number of false positives at various delay factors for our benchmarks (for both sizes 10/100). 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*. Unfortunately, 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 maximum gains from early compilation.

Figure 11 plots the ratio of the average performance improvement seen at various delay factors to the average ideal early compilation benefit available from Figure 3. 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 3 by expressing the appropriate false positive number in bold fonts. Figure 11 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, indicating that there is some potential for tuning the delay factors as well.

Figure 12 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.

5.3 Feasibility of Early Determination of Loop Iteration Counts

The success of our proposed online profiling technique depends on the ability of the VM to automatically determine the iteration bounds of loops. In this section, we explore the feasibility of determining the loop bounds (for 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 13 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. 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/dynamic analysis 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 comment in bold for Loop C), is feasible. In future work, we plan to study static/dynamic transformations to prepare Category C loops for automatic analysis during our profiling scheme.

On the other hand, loops in Figure 14 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-

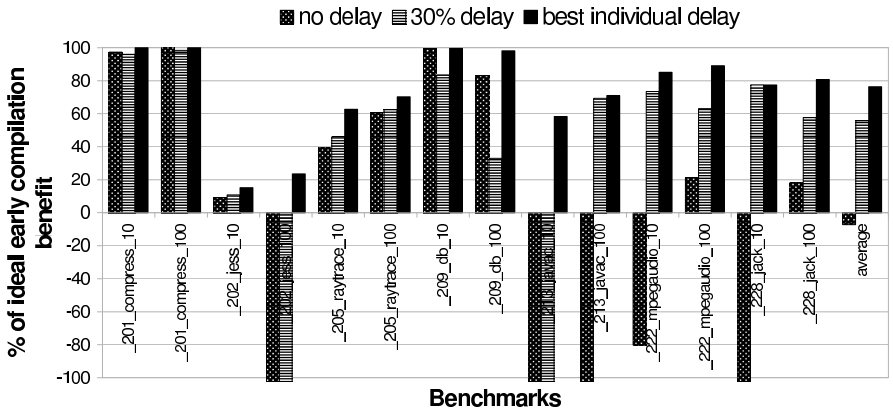

Figure 12. Individual benchmark performance results of delaying compilation to improve prediction accuracy, as a percentage of ideal early compilation benefit at selected delay factors

```
for(i=0 ; i<2500 ; i++)
{
    static_hot();
}
```
Category A

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

```
void method(List list){
    Iterator it = list.iterator(
    // n = list.size();
    while(it.hasNext()){
        el = (Element) it.next()
    }
}
```
Category C

Figure 13. Iteration bounds of some categories of loops can be accurately predicted early

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

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

Figure 14. Iteration bounds of some categories of loops are difficult to predict

Benchmark	Total Loops	Analyzable Loops			
		Category			% of Total
		A	B	C	
_201_compress	26	1	15	2	69.23
_202_jess	171	1	140	4	84.79
_205_raytrace	43	9	21	2	74.42
_209_db	21	2	12	1	71.42
_213_javac	237	3	108	41	64.13
_222_mpegaudio	77	23	43	2	88.31
_228_jack	89	2	29	37	76.40

Table 4. Statistics for Analyzable Loops

specific data structures, and category *E* loops are non-linearly dependent on input values. We term such loops as *non-analyzable* in this paper.

We performed a manual study using the Soot framework to find the percentage of analyzable loops in the SPECjvm98 benchmark programs. Table 4 lists the results of this study. For each benchmark, columns three, four, and five 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 5 and Figure 15 show the results of discarding the non-analyzable loops from the simulation runs, and correspond to the numbers presented earlier in Table 3 and Figure 11 respectively. Not surprisingly, discarding non-analyzable loops delays the early detection of hot methods in some cases. However, we found that eliminating the non-analyzable loops also has the unexpected side-effect of reducing the number of false positives. The combination of reduced false positives and delayed detection of *actual* hot methods, improves performance over the all-loops configuration of figure 12(a) for the lowest delay factor, but results in some degradation for the remaining factors. The most prominent reduction in the performance gain is witnessed for the *_201_compress* benchmarks, in which case the improvement over the default VM performance drops to about half. Overall, we measured an average improvement of 66% of ideal offline early compilation benefit considering the *best* delay factor for each benchmark.

5.4 Implementation Cost

In order to remain completely transparent to the user, the online profiling approach described in this paper 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

Benchmark	False Positives at Delay Factors (% of baseline threshold)										
	0%	1%	3%	7%	10%	20%	30%	40%	50%	70%	90%
_201_compress	2/3	2/1	**0**/**0**	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
_202_jess	3/25	3/17	3/13	3/**11**	1/9	**0**/6	0/6	0/4	0/4	0/4	0/0
_205_raytrace	18/7	5/5	**4**/**4**	4/4	4/4	4/4	2/2	2/2	2/2	0/0	0/0
_209_db	**1**/**3**	1/3	1/2	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
_213_javac	55/207	54/126	51/105	36/84	36/71	35/15	30/**0**	23/0	13/0	**5**/0	0/0
_222_mpegaudio	20/44	13/9	7/**5**	4/1	**2**/1	1/1	1/1	1/1	0/1	0/1	0/0
_228_jack	9/14	7/10	**0**/2	0/**0**	0/0	0/0	0/0	0/0	0/0	0/0	0/0

Table 5. False positives at different delay factors (for both input sizes 10/100) after removing non-analyzable loops

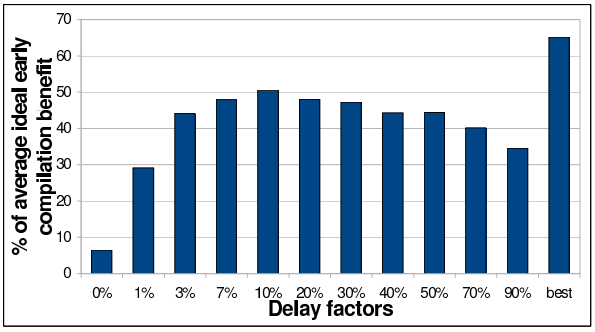

Figure 15. Performance benefit with analyzable loops only

pass will replace the static analysis and classfile annotation phase currently performed by Soot during our trace-file simulation (Step 1 of Section 5.1.1). 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 5.1.1), 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 5.1.2) 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 [Arnold et al. 2000]. 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 15 by 3.4%.

6. Future Work

There are a variety of enhancements that we plan to make in the future. First, 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 5.4. Second, we are also working to expand our benchmark set to include newer and greater number of programs. In fact, preliminary results on the SPECjvm2008 startup benchmarks show an ideal performance improvement of over 10% due to early compilation of hot methods. Third, 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. Fourth, 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.

7. Conclusions

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.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions.

References

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.

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.

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

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.

Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (ajit) complilation system. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 142–151, New York, NY, USA, 1999. ACM. ISBN 1-58113-161-5. doi: http://doi.acm.org/10.1145/304065.304115.

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.

D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. De 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.

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.

MichałCierniak, Guei-Yuan Lueh, and James 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. ISBN 1-58113-199-2.

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.

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.

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.

M. X. Goemans. Advanced algorithms. Technical Report MIT/LCS/RSS-27, 1994. URL `citeseer.ist.psu.edu/article/goemans94advanced.html`.

S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

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.

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

Tim Harris. Controlling run-time compilation. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, pages 75–84, December 1998.

Kim Hazelwood and David 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. ISBN 0-7695-1913-X.

U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18 (4):355–400, 1996. ISSN 0164-0925.

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.

Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.

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.

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.

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

T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 69–78, Washington, DC, USA, 2003.

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.

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.

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.

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.

Michael Paleczny, Christopher Vick, and Cliff 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.

Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, 1990. ISSN 0362-1340.

Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie J. Hendren, and Clark 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. ISBN 3-540-41861-X.

SPEC98. Specjvm98 benchmarks. http://www.spec.org/jvm98/.

Kaffe Developer Team. Kaffe java virtual machine. http://www.kaffe.org/, September 19 2007.

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.