Use of profilers for studying Java dynamic optimizations

Kevin Arhelger, Fernando Trinciante, Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris MN, 56267
arhel005@umn.edu, trinc002@umn.edu, elenam@umn.edu

Abstract

Java differs from many common programming languages in that Java programs are first compiled to platform-independent bytecode. Java bytecode is run by a program called the Java Virtual Machine (JVM). Because of this approach, Java programs are often optimized dynamically (i.e. at run-time) by the JVM. A just-in-time compiler (JIT) is a part of the JVM that performs dynamic optimizations.
Our research goal is to be able to detect and study dynamic optimizations performed by a JIT using a profiler. A profiler is a programming tool that can track the performance of another program. A challenge for use of profilers for Java is a possible interaction between the profiler and the JVM, since the two programs are running at the same time. We show that profiling a Java program may disable some dynamic optimizations as a side effect in order to record information about those methods.
In this paper we examine interactions between a profiler and dynamic optimizations by studying the information collected by the profiler and program run-time measurements without and without profiling. We use Java HotSpot™JVM by Sun Microsystems as a JVM with an embedded JIT and HPROF profiling tool. The paper details the testing methodology and presents the results and the conclusions.

1 Introduction

Java programming language has a two-phase compilation model: a Java program is first compiled to platform-independent bytecode instructions and then the bytecode is interpreted by the Java Virtual Machine (JVM). JVM has the ability to load and reload bytecode dynamically. This leads to the need to perform program optimizations dynamically as the program is being executed. Section 2 gives a detailed overview of Java compilation model and HotSpot JVM.
The paper investigates the usefulness of profilers for detecting dynamic optimizations such as method inlining. We describe common approaches to profiling Java such as *bytecode*

injection (see section 2.6). We discuss the challenges of determining whether interactions between the profiler and the JVM executing the program can change or disable optimizations performed by the dynamic compilers integrated into a JVM. Section 3 presents two main examples of optimizations studied using a profiler: one of dead code elimination, and the other one of method inlining. Using these examples we show that profilers may indeed disable dynamic optimizations. The paper stresses the complexity of Java dynamic compilation, the challenges of getting accurate information about its inner workings, and discusses methodology that we successfully used to observe dynamic behavior.

2 Background

2.1 Early implementations of Java

The Java programming language is utilized in a wide range of applications, from games, pervasive equipments, and e-business to mission-critical applications. One of the reasons for this is the ability to run the code independently from the operating system in use or the architecture of the hardware underneath it. All this is possible because Java is compiled into an intermediate language called *bytecode* and then executed on any system that has a Java execution environment, referred to as the *Java Virtual Machine (JVM)*.
Java language compilation is different from statically compiled languages such as C or C++. For example a static compiler will compile source code directly into machine code that will be executed on the target platform. Different hardware or platforms will require a new compilation of the original source code, making the code non-portable. In contrast the Java compiler converts the source code into portable bytecode that is compatible with any JVM.
At first, JVMs just interpreted bytecode, i.e. executing bytecode instructions one by one. This approach lacked in performance. The next generation of JVMs was created with the aim of increasing speed and performance without sacrificing portability.

2.2 Sun's HotSpot JVM and dynamic compilation.

Unlike statically compiled languages, in which the compiler does the optimizations, in Java they are performed at runtime, i.e. when the program is executed. This provides for maximum utilization of the specific hardware on which the program is being executed while still maintaining portability. Most modern JVMs are equipped with a *Just-in-time compiler (JIT)* that performs optimizations as the program is being run and may convert a portion or all of a program's bytecode to native code "on the fly". This process is called *dynamic compilation*.
HotSpot is the name of Sun's Java Virtual Machine. It is also defined as a functional design JVM, because based upon the platform configuration, it is capable of selecting a compiler (i.e client or server), a heap configuration, and a garbage collector that will produce the best results for most applications on the specific system.
Within HotSpot's execution process a few things happen in the background such as interpretation, profiling, and dynamic compilation. Upon taking a closer look at HotSpot

execution, we can see that rather then translating all the bytecode into native language before execution, HotSpot decides to first run as an interpreter and compile only the "hot" code. While execution is taking place, it gathers profiling data which is used to determine the "hot" sections of code that are executed frequently, then the compiler can spend more time optimizing the "hot" code. For example, the profiling data would allow the compiler to determine whether to inline a particular method call or not [8]. Since compilation is deferred until it is needed, it may occur more then once while the program is running.
Continuous recompilation is an aspect of the HotSpot approach in which compilation is not an all-or-nothing proposition. Once the path of the code is interpreted a number of times, it is then compiled into native machine code. However the JVM continues profiling and may re-compile the code again later with a much higher level of optimization if it decides the path is "hot" or further profiling data suggest possibilities for additional optimization. We can see continuous recompilation in action if we run the JVM with `-XX:+PrintCompilation` flag, which tells the compiler to print a message every time a method is compiled [3].
Since compilation is based on code profiling, methods are compiled only after a certain number of calls. This process is known as *bytecode warmup*, and the number of calls that triggers compilation is called the *compilation threshold*. The compilation threshold is unique for different modes of the HotSpot JVM (see section 2.3 below).

2.3 Client and server compilers of HotSpot JVM

As an added layer of complexity, HotSpot comes with two dynamic compilers, the *client* and *server* compilers. Correspondingly, there are two modes of HotSpot execution: the client and the server modes. The difference between the two modes is quite drastic. In essence they are two different JITs that interface to the same runtime system. The client mode is optimal for applications which need a fast startup time or small memory footprint. This is common for interactive applications such as GUIs. The server mode is optimal for applications where the overall performance is more important then a fast startup, such as server-side web applications. Other important differences between the client and the server modes are compilation policy, heap defaults, and inlining policy. According to Sun, client and server are present on the 32-bit JVM but only the server is on the 64-bit. However, a 64-bit architecture does not limit the system to a 64-bit JVM, since the 32-bit JVM is fully supported as well.
More specifically, on the Java 2 Platform, the client dynamic compiler has the following features [8]:

- Simple and fast 3 phase compiling.
- Platform-independent front end constructs a *high-level intermediate representation (HIR)* from the bytecodes.
- The HIR uses *static single assignment* (SSA) form to represent values. SSA is an intermediate representation in which each variable is assigned only once. SSA improves compiler performance because values of variables do not change so the code is easier to analyse[4].

- In the second phase, the platform-specific back end generates a *low-level intermediate representation (LIR)* from the HIR.
- The final phase performs register allocation on the LIR using a customized version of the linear scan algorithm [4].
- In this mode, the emphasis is placed on extracting and preserving as much possible information from the bytecode. For example, locality information and initial control flow graph are extracted, along with other information. This approach leads to reduced compilation time. Note that the client VM does only minimal inlining and no deoptimization. No deoptimization tells us that client mode makes emphasis on local optimizations [4].
- Default compilation threshold:$\sim$ 1500.

The server compiler comes with the following set of features [8]:

- It is tuned for the performance patterns of a typical server application.
- It is a high-end fully optimizing compiler.
- It uses an *advanced static single assignment (SSA)-based* intermediate representation.
- The optimizer performs all the classic optimizations, including dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, global value numbering, and global code motion.
- Features optimizations more specific to Java technology, such as null-reference check, range-check elimination (eliminating unneeded checks for an array index positioned within the array), and optimization of exception throwing paths [9].
- The register allocator is a global graph coloring allocator and makes full use of large register sets that are commonly found in RISC microprocessors.
- Default compilation threshold:$\sim$ 10000.

2.4 Dynamic optimizations

Below we discuss dynamic optimizations that are particularly relevant to our work:

- *Feedback-directed optimizations*: the server compiler performs extensive profiling of the program in the interpreter before compiling the Java bytecode to optimized machine code. This profiling data provides information about data types in use, hot paths through the code, and other properties. The compiler uses this information to more aggressively optimize the code. If one of the assumed properties of the code is violated at run time, the code is *deoptimized* (i.e. rolled back to the original bytecode) and later recompiled and reoptimized [8].

- *Deep inlining* and *inlining of potentially virtual calls*: method inlining combined with global analysis and *dynamic deoptimization* are used by both the client and server compilers to enable deep inlining (i.e inlining of methods which call other methods later in the path).

- *Fast instanceof/checkcast*: the Java HotSpot VM and compilers support a novel technique for accelerating the dynamic type checks reducing an overhead of object-oriented features.

2.5 Framework and Goals

Our goal in this research is to determine whether a profiler can be used to detect dynamic optimizations and measure their effects. Specifically, we are interested in studying programs that are highly intensive in method calls and not very intensive in memory allocation and garbage collection. Such programs come up in our work on Java optimizations that is beyond the scope of this paper (see e.g. [5]). In these programs the main benefits come from the following kinds of JIT optimizations:

- *method devirtualization*: replacing a dynamic method lookup along a class hierarchy by a direct jump to the method when the target of the call can be uniquely determined,

- *method inlining*: replacing a method call by a copy of the method's code when the call target can be uniquely determined (often follows method devirtualization),

- *dead code elimination*: removing unused portions of code.

Optimizations related to efficient heap allocation and garbage collection do not play a significant role in programs that we aim to study.
Unfortunately the HotSpot JVM does not have an easy way of monitoring the kinds of optimizations that we are interested in. Instead we have been using an indirect approach: we wrote programs that loop many times over code fragments that we were interested in and measured the programs' running time. By changing a program slightly and comparing the running time of different versions we were able to get an indirect evidence of whether an optimization has been performed. We also used flags that were available to control some optimizations in the HotSpot JVM. For instance, we have used the flag `-XX-Inline` that turns off inlining. If a program's running time noticeably increases with the flag then the program is likely to be affected by inlining in a run without the "no-inline" setting.
While there are ways to collect indirect evidence by measuring running times and setting JVM flags, these approaches do not allow a fine-grained analysis. For instance, even if it can be determined that a program benefits from inlining, it would still be unknown which of its methods are inlined. It would also be unclear whether other optimizations, such as dead code elimination, played any part in the running time change: inlining often triggers other optimizations by creating large contiguous code segments available for analysis.
Our goal in this research is to determine whether a profiler can be used to gain more direct information about dynamic (JIT) optimizations. For example, a profiler might be able to provide information about inlined methods or time spent on dynamic method lookup. Note,

however, that a profiler and a JVM are running at the same time, and it is possible that in order to collect the needed information a profiler may directly or indirectly disable some JIT optimizations.

2.6 Profilers

A **profiler** is a software tool that measures the time a program takes on each task it performs while it is executing. Profilers are used for a variety of programming languages. However, an additional challenge for a Java profiler is the interaction between the profiler and the JVM, since these two programs are running at the same time. In Java there are two primary methods of profiling code. They differ in the ways they interact with the JVM. We describe these approaches below.

The first approach is to periodically (normally around 200ms) query the JVM's execution stack and find the currently executing code. This is a very fast operation and results in little overhead to the execution time of the program. The profiler then outputs the most common occurrences that appeared in the query results. We found these results not to be useful as they do not show how long the code was executing, or how many times a certain function was executed.

```
TRACE 300051:
        ArrayListTest.functonTest(ArrayListTest.java:12)
        ArrayListTest.main(ArrayListTest.java:32)
TRACE 300037:
        java.lang.Integer.valueOf(Integer.java:585)
        ArrayListTest.functonTest(ArrayListTest.java:13)
        ArrayListTest.main(ArrayListTest.java:32)
```

Figure 1: Two sample traces from HPROF cpu sampling output

```
 CPU SAMPLES BEGIN (total = 797) Wed Dec 31 18:00:00 1969
rank   self  accum   count trace method
   1 79.55% 79.55%     634 300051 ArrayListTest.functonTest
   2 14.81% 94.35%     118 300037 java.lang.Integer.valueOf
```

Figure 2: Summary of samples from HPROF

Figure 2 shows that that out of 797 inspections on the JVM's stack, sample 300051 (see Figure 1) showed 634 times. While this may be useful in tracking down a problem area within the code, it is simply stating that 79.55% of out time was spent in a single method.

The second method utilizes bytecode injection: before each method or function is called, the profiler "injects" a small piece of code at the beginning and end of the method. The purpose of this injected code is to record the time at the beginning and end of the method and also counting how many times the method is called. HPROF, JProfiler and TPTP (see below) can operate in this manner.

Sun included new integration points for profilers and debuggers in *Java 5*. They called it the Java Virtual Machine Tool Interface or **JVMTI**. The new interface replaces both the older *Java Virtual Machine Profiling Interface* and the new *Java Virtual Machine Debug Interface* [7]. This allows anyone to write C and Java code to inspect the JVM while it is running. The interface allows for examining memory management, the heap, the current stack frame and also "Bytecode Instrumentation" (also know as "bytecode injection"). All the profilers we examined use this interface for gathering their profiling data.
Our criteria for choosing a profiler were:

- Able to run multiple times from a script to gather statistical data.
- Output profiling information into a parsable format for later analysis.
- Documentation and Accessibility. In order for other groups to use our results there must be good documentation. Also an expensive, commercial profiler would not be accessible to other research groups.

These are the profilers we examined for our research.

- **HPROF** is a simple profiler included with Sun's Java Development Kit (**JDK**) [6]. It uses the JVM's standard Java Tools interface. HPROF includes the basic features, heap profiling (the number and size of allocated objects), and CPU profiling (through traces and code injection). HPROF is well documented by Sun and outputs easily parsable text profiling data. For these reasons HPROF has been our main profiler.
- **JProfiler** is a commercial profiler developed by *ej-technologies*. While the main use of JProfiler is via its GUI, the command line interface was more applicable to our use. The GUI allows one to define triggers where different kinds of profiling occur. An example of a trigger would be: "start cpu profiling 30 seconds after the program starts". Another useful feature is its filtering capability, i.e. the ability to define what classes are not profiled. Also the results can be automatically exported, through the use of a trigger, into CSV, XML or HTML format. The JProfiler GUI can collect all the settings and export a command line script that can be used "offline" (i.e. not using the GUI). After evaluating a trial version of JProfiler our decision was that the results where similar to HPROF and the cost to continue using JProfiler was not justified.
- **TPTP** is the Eclipse projects profiler. Its main feature is its easy integration with the Eclipse IDE. It has a client/server design where the client can either be the Eclipse plugin or a command line program. Usage of TPTP outside of Eclipse was poorly documented. We were unable to easily configure the profiling settings from the command line. Also the default output of the profiler was in a binary format. While TPTP would be a good choice for those using the Eclipse environment, our need was to run multiple automated profiling sessions in a row without the overhead of an IDE.
- **OProfile** started out as a system-wide profiler for Linux. More recently the project has added a Java profiling extension (also using the JVMTI) that allows JITed bytecode to be profiled at the system level. Our operating system did not have any precompiled versions on this newer version of OProfile. Also the online documentation

for OProfile almost exclusively talks about the system level profiling, and only mentions the JIT support as an afterthought.

3 Optimizations, Challenges, And Methodology

3.1 Creating Optimizations

Our goal with each optimization was to create two similar examples. One example would perform a single, specific optimization. The other example would be similar, and with a small change prevent that specific optimization from occurring. We verify that this is occurring by checking the runtime differences on the optimized and unoptimized exampled. After we are certain that the optimization is indeed occurring, we run our tests with profiling enabled. By comparing the results we can ascertain if the profiler is indeed affecting the optimization.

3.1.1 Dead Code Elimination

An obvious test of a profiler's side effects is to test it on a dead code elimination example. The example program makes a call to a method `sum` that computes the sum of integers from 0 to 9,999 (see Figure 3). The loop in the `main` method of the program increments a variable `result` by the return value of a call to `sum()` (see Figures 4, 5).
Our two examples share the code for `sum` and the main loop. However, one triggers dead code elimination (Figure 4) by not printing the result at the end of execution. The other one (Figure 5) prints the result so the code must be executed. We based these two examples off of a developerWorks [1] article.

```
 private static int sum() {
    int sum = 0;
    for (int j = 0; j < 10 * 1000; j++) {
        sum += j;
    }
    return sum;
}
```

Figure 3: Common method `sum` for both examples

As expected on the Server VM (reflected in Figure 8), with profiling turned off, the Dead Code test completed in 0.14 seconds and the LiveCode Test in 5.86 seconds, indicating that the dead code elimination takes place. However, when the profiler was turned on, the Dead and LiveCode showed similar times. The Dead Code which originally completed in 0.14 seconds due to dead code elimination now completes in 10.4 seconds. Also, the LiveCode jumped from 5.86 to 10.6 seconds due to profiling overhead.
The results are easily seen by comparing the profiled result in Figure 6 to the profiled example in Figure 7. We believe this is due to the profiler injecting code into the sum

```
 public static void main(String[] args) {
    long t1 = System.nanoTime();

    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) {    // sole loop
        result += sum();
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = ");
}
```

Figure 4: Main method that results in dead code elimination

```
 public static void main(String[] args) {
    long t1 = System.nanoTime();

    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) {    // sole loop
        result += sum();
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = " + result);
}
```

Figure 5: Main method that prevents dead code elimination

function. Since the injected code has side effects, JIT can not ignore the sum function any longer.
We also observed that the Client VM compiler does not perform dead code elimination so the running times of the two examples without profiling were the same in the client mode.

3.1.2 Inlining

The next obvious optimization to examine is method inlining. Method inlining involves moving the body of the method directly to the area from which it is called. An example that can trigger inlining would follow this setup:

```
...
int counter = 0;
counter = foo(counter);
System.out.println(counter)
```

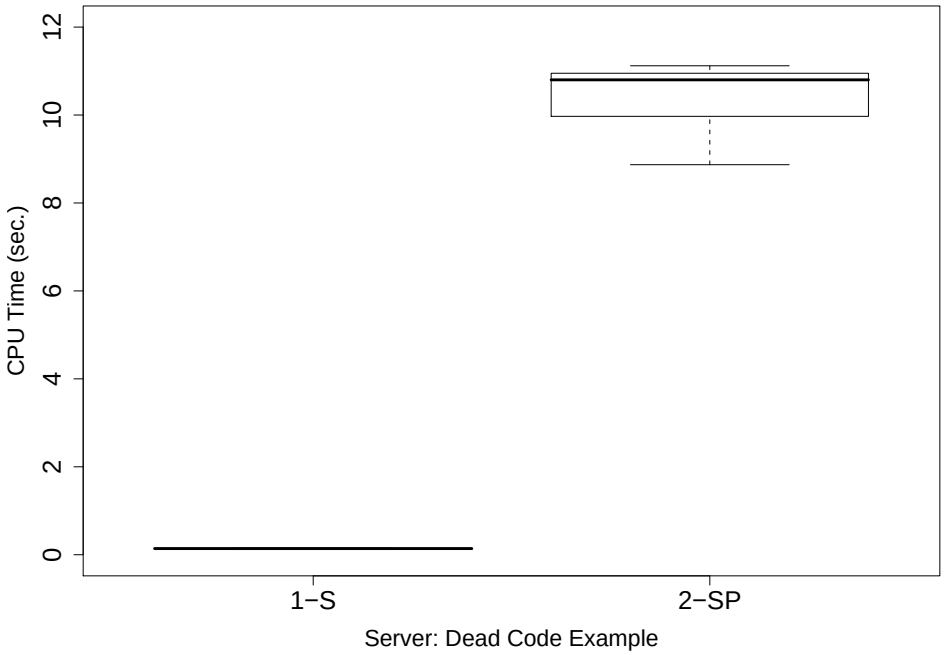

Figure 6: Dead Code on Server VM

```
...
int foo(int n) {
return n + 1;
}
```

The JVM may then dynamically optimize this code as follows.

```
...
int counter = 0;
counter = (counter + 1);
System.out.println(counter);
...
```

This is a simple example of inlining. This optimization is advantageous because it removes the overhead of adding the method to the stack or returning from the method. Chapter 8, Performance Features and Tools[2] suggested that inlining was both easy to trigger and showed definite results within the profiling output. Their results showed that inlining would prevent the method from showing in the profiling output. Also an AMD developer suggested a method for detecting inlined methods that do not show up in a profiler's output [10].
Before profiling we needed to be able to determine whether or not a specific method was inlined. We attempted to use the examples from the Chapter 8 with and without the

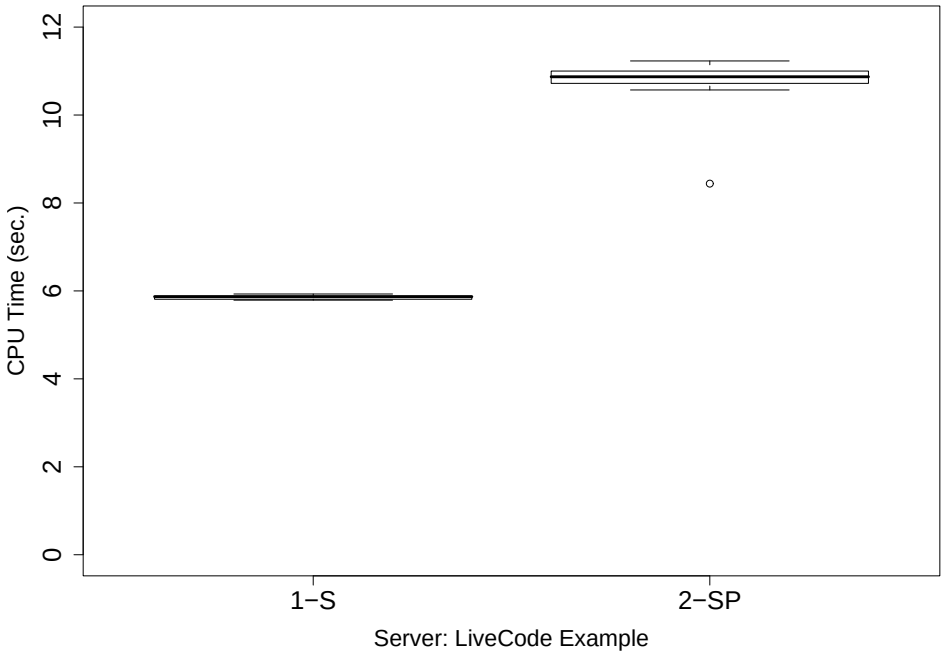

Figure 7: LiveCode Server

	Dead Code	LiveCode	DeadCode 10X	LiveCode 10X
Not Profiled	.14	5.86	.145	68.82
Profiled	10.41	10.65	103.49	102.81

Figure 8: Dead Code Elimination Results

`-XX:-Inline` JVM flag to determine if they were in fact being inlined. The results from this test were inconclusive, increasing the number of loops by a factor of 1000 also did not change our results. Both samples ran in very similar times, and using the `-XX:-Inline` flag almost exactly doubled both runtimes. Profiling the samples also showed the same increase in both test cases.
Since these results are inconclusive we have not included any figures showing the changes. The *Performance Features and Tools* was using the latest release of Java (1.4) in 1999. Our hypothesis is that improvements in the JVM's JIT have made this example obsolete.

3.2 Challenges

3.2.1 Slow Method Calls

One of our examples used a small integer Array and looped through the elements, adding one to each. Each element was incremented 1,000,000 times and finally a random element from the array was returned (to prevent dead code elimination). We then modified the

original to use a method call to increment the array element instead.
The original increment line (the Fast example) in the loop is `array[y] += 1;`. This was changed in the Slow example to `addOne(y);`. The "addOne" method is as follows.

```
private static void addOne(int n) {
    array[n] += 1;
}
```

We ran both Fast and Slow with all combinations of the client and server VM, and profiled, not profiled.

	Fast Client	Slow Client	Fast Server	Slow Server
Normal	.10	.10	.12	.12
Profiled	.25	107.96	.24	100.74

As can be seen from the table, when profiling is turned on a drastic increase in times happens.
This was not an isolated example. We developed a pair of tests, where one used an Array and the other one an ArrayList. Everything else in the samples was identical. An even larger increase was seen as calling "set" on an array list: in addition to the cast from `int` to `Integer`, the sample program calls a half a dozen methods, all of which add to the time the program running time.
We found that both HPROF and JProfiler added this large overhead to method calls when cpu profiling was enabled. We believe this to be caused by the overhead of the injected bytecode.

3.2.2 Profiled Code May Be Faster Without Method Calls

When using our simple Array example, we found that by enabling the profiler and increasing the number of iterations, we could arrive at a result where the profiled code was completing faster then the non-profiled. This change was most visible in the Server VM, but also noticeable (after more iterations) on the Client VM. Our conclusion to this was that the profiler a slight overhead to decrease the number of iterations till the JVM's optimization threshold was met. We have only observed this behavior when using code with very few method calls.

3.3 Methodology

We have adopted a similar methodology that has been used by other research groups working with Professor Machkasova. All code and test results are stored inside a cvs repository. This allows for the test to be easily checked out and changes to the code recorded. All tests are run on the same machine in /tmp to ensure that network problems do not interfere with the tests

Our Test Machine.
AMD Athlon™64 Processor 3200+
512MB DDR RAM
Fedora Core 7
Kernel: 2.6.23.17-88.fc7 SMP i686
Java Version: Sun JDK 1.6.0_04
Time Binary: GNU time 1.7
glibc version: 2.6

Out test platform uses ruby scripts to set up the environment, run the tests, record runtimes, and auto-generate R graphs from the runtimes. Runtimes are measured with the /usr/bin/time binary. Our scripts use `time -f%U command` piped to a file to calculate the program runtime. We have previously found that using user time is the best approximation of a java programs runtime since "real" time (i.e. wall clock) may be affected by anything else the test machine happens to do at the same time.
An example of a configuration file

```
Conf = {
    'source_file' => 'DeadCodeExample.java',
    'java' => '/usr/java/jdk1.6.0_04/bin/java',
    'javac' => '/usr/java/jdk1.6.0_04/bin/javac',
    'compile' => true,
    'run_name' => 'DeadCodeExample_10_runs',
    'num_runs' => 10,
    'client' => true,
    'server' => true,
    'profile' => true,
    'profile_args' => '-agentlib:hprof=cpu=times',
}
```

This example configuration file would use the java source file in the local directory named DeadCodeExample.java, compile with the defined javac, run four "batches". The batches would be 10 each in client VM, server VM, profiled in client VM and profiled in server VM. The results will be in the subdirectory DeadCodeExample_10_runs. We save the profiler output, program output (including the result of the time command). The full command line is also logged. After the batches complete, it generates R graphs and summary statistics.

4 Results And Conclusions

Our primary goal was to determine if a certain optimization was being performed simply by consulting the output of a profiler. What have determined thus far is because the profiler modifies the bytecode, determining if an optimization is being performed is not easy.

Our results show that a profiler performing bytecode injection does prevent bytecode injection from happening as seen in Section3.1.1 We showed that because of the bytecode injection the code cannot be optimized.
Our results with inlining (Section 3.1.2) were inconclusive. We failed to successfully find two similar examples where one was inlined and the other was not. All the examples we tried exhibited similar behavior even with inlining turned off at the JVM level.
As a more general note, we documented that a bytecode-injecting compiler may add side effects to code such as long runtimes (Section 3.2.1) and in some cases it may even cause it to run faster (Section 3.2.2.)

5 Future Work

Our example of a dead-code elimination clearly shows that profilers may change dynamic optimization of a program. Our goal is to study interactions between profilers and the HotSpot JVM further and find out when profiler information **is** reliable and informative. We plan to combine the study of profiler information not only with time measurements, but also with other means of detecting dynamic optimizations. A promising direction is to use HotSpot command-line options (or *flags*). HotSpot has a variety of flags to fine-tune the optimizations (such as the flag that turns off inlining) and to monitor dynamic compilation (such as a newly introduced flag `-XX+LogCompilation`). By combining different ways of monitoring dynamic compilation we hope to be able to obtain accurate and reliable information about dynamic optimizations. Such methodology will be helpful not only in our research on Java optimization, but also for a community of Java researchers and developers in general.

References

[1] BOYER, B. Robust java benchmarking. *developerWorks, IBM, ibm.com* (2008).

[2] CALVIN AUSTIN, M. P. *Advanced Programming for the Java 2 Platform*. Addison Wesley Longman, 2000.

[3] GOETZ, B. Java theory and practice: Dynamic compilation and performance measurement. *IBM Developer Works* (2004-2007), 1–8.

[4] KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. Design of the java hotspot™client compiler for java 6. *ACM Trans. Archit. Code Optim. 5*, 1 (2008), 1–32.

[5] MAYFIELD, E., ROTH, J. K., SELIFONOV, D., DAHLBERG, N., AND MACHKASOVA, E. Optimizing java programs using generic types. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* (New York, NY, USA, 2007), ACM, pp. 829–830.

[6] O'HAIR, K. Hprof: A heap/cpu profiling tool in j2se 5.0. *Sun Microsystems, java.sun.com* (2004).

[7] O'HAIR, K. The jvmpi transition to jvmti. *Sun Microsystems, java.sun.com* (2004).

[8] SUN DEVELOPER NETWORK. The java hotspot performance engine architecture. *Sun Microsystem* (2007).

[9] SUN DEVELOPER NETWORK. The java hotspot™server vm. *Sun Microsystem* (2008).

[10] VENKATACHALAM, V. Inlining information hidden in the hotspot jvm. *AMD Developer Central, amd.com* (2009).