The Observer Effect of Profiling on Dynamic Java Optimizations

Elena Machkasova
University of Minnesota, Morris
elenam@morris.umn.edu

Kevin Arhelger
University of Minnesota, Morris
arhel005@morris.umn.edu

Fernando Trinciante
University of Minnesota, Morris
trinc002@morris.umn.edu

Abstract

We show that the bytecode injection approach used in common Java profilers, such as HPROF and JProfiler, disables some program optimizations that are performed when the same program is running without a profiler. This behavior is present in both the client and the server mode of the HotSpot JVM.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization,run-time environments; D.2.8 [*Software Engineering*]: Metric—performance measures

General Terms Measurement, Performance

Keywords Java,HotSpot,JVM, profiler,HPROF,inlining,dead code elimination

1. Introduction

In a traditional implementation Java™programs are compiled to bytecodes and then ran by a Java Virtual Machine (JVM). Because of the need to dynamically load and reload Java classes, most Java optimizations are performed dynamically by Just-in-Time compilers (JITs). Most modern JITs employ an adaptive compilation approaches when only frequently executed code (so-called *hot spots*) gets compiled to native code and/or optimized. A JIT-equipped JVM is a sophisticated system that achieves significant speed improvements. However, detecting specific program optimizations in this complex run-time environment is very challenging.

A common tool for monitoring a program performance is a profiler. Many profilers for Java, such as HPROF and JProfiler, use a technique called *bytecode injection* (see section 2.2) to keep track of methods being executed. While profilers are extremely useful for monitoring memory usage, garbage collection, and other important aspects of run-time behavior, we show that they also have an "observer effect" on programs: a mere use of a profiler may disable some program optimizations, such as dead code elimination and method inlining, that are performed when the program is running without profiling.

We use small Java sample programs in which some parts of code trigger the dynamic optimizations in question. To prove that the optimizations take place, we time the programs and compare them with "hand-optimized" versions of the programs Then we run the same programs with a profiler and study its output and timing. We show that while the optimizations take place for a non-profiled program, they are not happening when the program is being profiled. Our observations are also applicable to other bytecode-based languages, such as JRuby and Jython, and to more languages if their profilers employ a similar technique.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

2. Background

In this project we used Java HotSpot™JVM by Sun Microsystems in both the *client* and the *server* mode [3]. We used *HPROF* [2] and *JProfiler* by ej-technologies.

2.1 Overview of HotSpot optimizations

The key feature of HotSpot JVM is that it starts each method by just interpreting its bytecodes. It later compiles to native code and/or optimizes only those methods that are executed over a specified number of times (known as *compilation threshold*). By default this threshold is 1500 in the client mode and 10000 in the server mode. A method needs to get over the threshold to get all benefits of dynamic compilation. Executing methods enough times to get over the threshold is called *JVM warmup*. In our examples the large number of method calls guarantees that the threshold is reached.

2.2 Overview of Java Profiling

A profiler obtains some method execution information by periodically sampling the stack. However, this information is inaccurate since it does not detect what happens in-between the sample points. An alternative *bytecode injection* approach inserts a small sequence of bytecode instructions at the beginning and end of each method to record the time spent in the method. HPROF and JProfiler use this approach.

run	Complex (S)	Easy (S)	Hand (S)	Complex (C)	Easy (C)	Hand (C)
Unix	0.282	0.284	0.282	6.446	6.45	6.456
Unix, -XX:-Inline	11.92	14.06	0.28	29.878	18.174	6.372
Unix, hprof (fewer loops)	4.634	4.876	0.176	4.92	4.64	0.204
Windows	0.43	0.474	0.45	7.982	8.05	7.972
Windows, -XX:-Inline	21.578	20.706	0.422	27.048	26.992	7.864
Windows, hprof (fewer loops)	5.882	5.268	0.244	5.482	5.242	0.252

Table 1. Mean runtimes for inlining (in seconds): server (S) and client (C)

3. Tests and Results

Detecting program optimizations is challenging since very little run-time information is available from a JVM.

We tested our sample programs in two different environments:

• Linux workstation with AMD Athlon™64 Processor running Fedora Core 7.

• Lenovo Thinkpad T42p with Intel Pentium M™Processor running Windows XP SP3.

We used Sun JDK 1.6.0_04. Our sample programs repeat the method or the code that we are studying a very large number of times in a loop (2147483647 for inlining examples) to produce total running times of several seconds. This makes differences between different running times clearly observable and JVM startup and warmup times insignificant. We repeat all tests 6 times, drop the first run since it takes extra time for memory allocation for the JVM, and record the mean of the remaining 5 runs.

When using a profiler, we had to reduce the number of loops because of the timing overhead of a profiler. Thus the profiled results are comparable to each other but not comparable to non-profiled results in terms of absolute times.

We wrote three small programs that perform the same task: repeatedly increment a variable. Two of them call a method to accomplish this task; they differ in the complexity level of the method. `EasyInline` has a simple method `return1` that just returns 1. It is called in a loop to increment the instance variable `counter`:

```
for(int i=0;i<2147483647;i++)
    counter += return1();
```

`ComplexInline` has a more convoluted method

```
public int addCount(int add) {
    add=add+1; return add; }
```

to accomplishes the same task; it is called in a similar loop as in `EasyInline`. `HandInline` has the functionality of the method hand-inlined directly into the loop:

```
for(int i=0;i<2147483647;i++) counter++;
```

The test results are summarized in table 2.1. While the absolute times differ for the Linux workstation and for Windows system, the pattern is the same. When no flags are specified, the three examples run in the same time. The two programs that call a method have the same runtime as the hand-inlined program, indicating that the method is inlined. This is confirmed by running the same three examples with `-XX:-Inline` flag to tunr off JIT inlining: the hand-inlined version does not change its running time, but the other two programs increase their time drastically.

When the programs are profiled, however, the methods that are supposed to be inlined show up in the profiling log, e.g. `HPROF` log for `EasyInline` (client mode, Linux):

```
1 60.31% 60.31%  ...  EasyInline.method1
2 38.72% 99.03%  ...  EasyInline.return1
```

`method1` is the method that contains the main loop. Here the first percentage value is the percent of time the method is being executed, and the second number is the combined percent of the top methods up to the given one. The other profiling logs also show that the method that was supposed to be inlined is still in the second place in the log. Note that the profiled runs are much faster for the hand-inlined version than for the other two (see table 2.1), which is another indication that inlining does not happen. JProfiler results have the same pattern as `HPROF`. We also observed a similar effect of profiling for dead code elimination, see [1].

4. Conclusions and Future Work

We have shown that bytecode injection in a profiler has an "observer effect", i.e. it changes the performance of the program simply by monitoring its behavior. This has significant impilcations for software developers who use profilers for performance tuning. Alternative methods for detecting the optimizations inlclude JVM flags that may display some relevant information, such as `LogCompilation` added in Java 1.6. However, their usefulness needs to be checked, and they require `Unlock Diagnostics` option that may itself have an "observer effect".

References

[1] K. Arhelger, F. Trinciante, and E. Machkasova. Use of profilers for studying java dynamic optimizations. *Proceedings of Midwest Instruction and Computing Symposium (MICS)*, 2009.

[2] K. O'Hair. HPROF: A heap/cpu profiling tool in j2se 5.0. *Sun Microsystems, java.sun.com*, 2004.

[3] Sun Developer Network. The Java HotSpot performance engine architecture. *Sun Microsystems*, 2007.