Code Profiling in Java

A rationale for Profiling and overview of techniques.

Andrew Wilcox
Senior Architect
MentorGen LLC

Introduction

This article will discuss the reasons why Java developers would need to profile their code and will dispel some common myths about code optimization. It will then discuss how profilers work and how they can be used to optimize code. We will look at the limitations of many profiling strategies and how interactive profiling can be used to get a better picture of how Java code will perform in a production environment. We'll look at using agents in Java5™ to build a simple profiler.

Why profile?

Profiling is a performance measurement technique. If your application is slow, it only makes sense to measure where the code is spending most of its time and attempt to make that code more efficient. However, if an application is having performance problems, most developers either go through the code to look for local inefficiencies or will go directly to the part of the code that they suspect is causing the problem. Both of these tactics are flawed.

Local optimization is the act of going through the code, line by line, to look for known inefficiencies. In the Java world there has been a lot said about things to avoid: avoid I/O, avoid String manipulation, avoid unnecessary object allocation, etc. From the larger discipline of algorithm analysis, we know how to quantify the efficiency of the algorithms that are used. But these are local optimizations and may not actually make the application in question run any faster. Most experienced developers have been on projects where performance was an issue, and someone has spent days or weeks optimizing suspect code only to discover that the effort yielded little or no improvement.

In complex applications, you need to look at the performance of the code as a whole and not the small section of code that you think is the culprit. You can optimize a piece of code all that you want, but if that code only represents 2% of the total time of a particular case, you have no hope of significantly improving the performance of you code. There are people who think that a 2% improvement here and a 5% improvement here will add up to a signification increase in performance. I call this the nickel and dime approach. It is almost always less successful than going after the big performance offenders.

The other often-employed tactic to improve performance is to go after that part of the system that the developer has "always suspected" was slow. I call this the Intuition Approach and in my experience it is almost always wrong. Even people who regularly profile code tend to be wrong about 90% of the time. This is because modern Java applications are complex. There is a lot going on and it is just not possible to be able to factor everything in and be able to guess where the performance bottleneck is. Adding to the complexity is the number of third party libraries that are used. Since developers aren't normally aware of the implementation details of these libraries, they do not know how expensive one particular call might be.

Both the Local Optimization Technique and the Intuition Technique cannot be relied on to help us optimize Java applications. We need an approach that will objectively measure application performance at every level of code to give us an accurate picture of what needs to be fixed in order to improve performance.

How to profile

In the Java world there are two major methods for measuring the performance of an application. The first is our good friend `System.currentTimeMillis()`. The other is using automated code profilers like `hprof` (the profiler that ships with the JDK).

Using `System.currentTimeMillis()` is probably the most common technique used for measuring the performance of a section of code. The basic approach is to measure the current clock time at the beginning of a routine and then at the end of the routine. The difference between the two is the time spent in the routine. This approach is simple and accurate. However, there are two big downsides to this approach. The first is that it is a manual process. The developer determines what will be measured, adds the code, recompiles, redeploys if necessary, runs some test and then gathers and analyses the results. When the profiling process is done all the code needs to be cleaned up. If profiling needs to be done again in the future, the code will need to be changed again and this process repeated.

The other problem with this approach is that it only measures performance on one level of code. The execution of a program or a particular section of code can be thought of as a tree structure where each node represents a method that is being executed. Measuring the performance of a single node doesn't tell you anything about the performance of sibling nodes nor does it necessarily give you a clear picture of the performance of child nodes. This is generally dealt with on an ad-hoc basis. In other words, if the developer decides that the performance of child nodes needs to be measured, code is added to do so. This can be time consuming, particularly if the build and deploy cycle is long or if the code being measured takes a long time to execute.

The other method of performance measurement is to use a code profiler like `hprof` (the profiler that ships with the JDK). A profiler avoids the problems associated with ad-hoc performance measurement. No special code needs to be added in order for the profiler to work. In the case of `hprof`, you don't even need to recompile your code or run it through a special post processor. In addition, `hprof` gathers timing information for every method call, not just ones at a certain point in the code. This gives the developer a more comprehensive view of how the code is performing.

Limitations of current approaches

So `hprof` does everything we need from a profiler, right? Well, not exactly. While `hprof` is a good tool, it does have a few limitations. There are a number of areas in which `hprof` falls short of what is needed to profile modern Java applications.

The biggest pitfall to using `hprof` is that it is slow. How slow? It's safe to say that it will make your code run at least 20 times slower on average. An ELT (extract, transform and load) routine that normally takes an hour to run could take almost a day to profile! Not only is it hard to be patient when waiting for results, this slowness can actually skew the results. Take for example a program that does a lot of I/O. Since the I/O is performed by the operating system, the profiler does not slow it down. From a profiling standpoint, the I/O will appear to run 20 times faster (with respect to the program) than it actually does! This means that `hprof` will not always give you an accurate picture of the true performance of an application.

Another pitfall of `hprof` has to do with how Java programs are loaded and run. Java programs are made up of classes. Unlike statically linked languages like C or C++, the classes in a Java program are linked at runtime rather than compile time. This means that when a Java program is run, a class is usually not loaded by the JVM until the first time it's referenced. If you are trying to measure the performance of a routine and the classes that are used by the routine have not been loaded by the JVM, part of the time you are measuring is the class loading time, not the application performance. Since classloading is done only once, you probably don't want to include it when measuring the performance of long-lived applications.

Yet another issue has to do with how Java classes are compiled. Java source files are compiled into byte code by the compiler. At runtime, the Java virtual machine will compile the byte code into native code so that the application will run faster. This is called Just In Time (JIT) compilation. Early JITs compiled byte code to native code when classes where class loaded. This resulted in static compilation which is very similar to how C and C++ programs are compiled. Modern JITs use a dynamic compiler which observers how the code actually runs before it generates native code. This observation allows the JIT to make "smart" optimizations and the resulting code can have better performance than statically compiled code. The difficulty with this is that you want to measure the performance of the code after the JIT has compiled optimized native code. For example, if the code in question is part of a long-lived application, like a web application, the first time the code is executed will be the slowest, since it is interpreted byte code rather than optimized native code. In addition, the classloader might be loaded classes which slows things down even more. The code might need to be exercised a number of times before the JIT has generated optimized native code for all of the classes. In the case of long-lived applications, you don't want to measure the performance of the code until the classloader and the JIT have done their work. Profilers like `hprof` don't allow you to easily to this.

Things get even more complicated when your code is running in an application server or servlet engine. Profilers like `hprof` profile at the virtual machine level. They start when the VM starts and run until the VM exists. This can be a problem since you really don't want to profile the servlet engine's startup and shutdown. Both only happen once, the users don't see it, and there is little you can do to change the way that it performs.

What is needed is a profiler that doesn't have too much overhead and can be started and stopped at will. That is, a lightweight interactive profiler.

Other Annoyances

There are a couple of other annoyances with profilers that are not necessarily limitations but it would be nice if we didn't have to deal with. These are the inability to filter out classes or packages from the profile, the need for native components and the fact that some solutions are closed source or have commercial licenses.

You'll run into the first annoyance when you profile large pieces of code. Profilers like `hprof` show you every single method call. This means there is a lot of information to sort through. Some of these calls are in third part classes that cannot be modified directly. While sometimes it is informative to see what is happening in third party libraries, more often than not, you only want to see code that you can change. What would be really nice is the ability to filter out classes or packages that you don't want to see. This is not to say that you'd want to filter out their execution time – that would be counter-productive. But it would be nice not to have to wade through a whole bunch of nonessential information.

Most profilers like `hprof` require native components. This is primarily because Sun's profiling interface, JVMPI (Java Virtual Machine Profiling Interface), requires the use of JNI (i.e., native components). This means that to some extent, profilers are platform dependent. It would be nice to have a 100% Java way of profiling Java code.

Another annoyance with many profilers is that they require a commercial license. Most developers who work in I.T. know how difficult it can be to get your company to pay for even inexpensive tools. Many times an evaluation process is also needed. One of the great advantages to Open Source software is that it can simply be downloaded and used without a lot of hassle. If you don't end up liking it or using it, at least you don't have to jump through a lot of hoops to find that out. This approach has been instrumentation to Java's success.

A New Approach to Profiling

Based on what's been discussed so far, this is what an ideal profiler would look like:

- Interactive, that is, the ability to be turned on an off interactively at run time. This would allow the developer to make multiple measurements at run time (rather than the single measurement that most profilers give).
- 100% Java. What's lacking in most profilers it that they require native components, which contradicts Java's "Write once" philosophy.
- Open Source. This gets us around many of the licensing issues that were discussed previously. Another advantage to an Open Source profiler that is

actually written in java is that it would be easier for 3rd parties to extend to meet their special needs.

- Lightweight. A profiler needs to be reasonably fast in order to capture accurate performance data.
- Filter by class or package. Rather than having to wade through a whole bunch of irrelevant data, you need to be able to filter out classes and packages that you don't want to see in the output. In addition, it would be ideal if filtered classes weren't measured at all, rather than just having the results filtered out after the fact.
- Performance timings. An ideal profiler would keep track of the overhead involved with profiling. This would allow it to factor out this time and would give the developer an idea of the actual amount of time that the code took to execute.

Using the agent interface and aspect-oriented techniques to build a profiler.

The basis of a solution

As I mentioned before, there are two approaches to profiling, using a profiler or using our old friend `System.currentTimeMillis()`. A very simple profiler could be built by outputting the system time at the beginning and end of each method. All you have to do is send this to `System.out.println()` and you're in business. Of course, the biggest problem with this is having to manually instrument the beginnings and endings of each method. Does this sound like a familiar problem? Anyone who is working with aspects should realize that the situation I'm describing is a perfect candidate for using aspects. However, there are many projects that still aren't using aspects. Can we use aspect-oriented techniques to build a profiler without requiring an aspect-oriented tool?

The javaagent

The Java5™ VM introduced an option, `-javaagent`, that allows developers to write code (agents) that hook into the VM's classloader and modify classfiles as they are loaded. This seems like to perfect tool for what we're trying to do; it will allow us to add a profiling aspect to Java classes as they are classloaded. To make this happen we'll need to look into how to write a java agent and we'll also need a way to manipulate the byte code so that we can add code for our profiler.

The javaagent VM option is, unfortunately, very sparsely documented. To make things more difficult, there aren't many books on the topic – no "Java Agents for Dummies" or "Teach yourself to write Java Agents in 21 days." The best source of information I've been able to find is an article called "Instrumentation: Modify Applications with Java 5 Class File Transformations[1]"). The basic idea behind an agent is that, as the JVM loads a class, the agent is given the opportunity to modify the classes' byte code, which is exactly

[1] "Instrumentation: Modify Applications with Java 5 Class File Transformations" by R. J. Lorimer (http://www.javalobby.org/java/forums/t19309.html)

what aspect-oriented frameworks need. For create an agent, there are only two requirements. First is that you implement the `java.lang.instrument.ClassFileTransformer` interface, which only has one method:

```
      public byte[] transform(ClassLoader loader, String className,
            Class<?> classBeingRedefined,
            ProtectionDomain protectionDomain,
            byte[] classfileBuffer)
            throws IllegalClassFormatException;
```

The other is to create a “premain” class. This is very similar to a class with the standard “main” method, but instead you use the following signature:

```
public static void premain(String args, Instrumentation inst)
```

Getting your feet wet

The best way to get started with something new is to get a very simple program running, rather than writing something large and complicated that has lots of places for potential problems. So let’s start by writing an agent that just prints out the name of a class when it’s class loaded. This would be very similar to the way that the `–verbose:class` VM option works. As you can see from the code in figure 1, what we’re doing is quite simple and only requires only few lines of code:

```
public class Main {
      public static void premain(String args, Instrumentation inst){
            inst.addTransformer(new Transformer());
      }
}
class Transformer implements ClassFileTransformer {
      public byte[] transform(ClassLoader loader, String className,
            Class<?> classBeingRedefined, ProtectionDomain
            protectionDomain, byte[] classfileBuffer)
            throws IllegalClassFormatException {

            System.out.println(className);
            return classfileBuffer;
      }
}
```

Figure 1 – a simple program that acts like the –verbose:class VM option.

The only other thing we’ll need is a manifest file. This instructs the VM’s agent loader which ‘premain’ to use (note that the format of this file, including spaces are line feeds is very important – see R. J. Lorimer’s article for more information).

```
Manifest-Version: 1.0
Premain-Class: sample.verboseclass.Main
```

Figure 2 – the manifest file.

What else is needed:

1. *Go through building the jar file*
2. *Create an example class that we can run our program against.*
3. *Go through running this on the command line*
4. *Side bar:*
    a. *How all of this stuff works with the various classpaths (application, extension, bootstrap).*
    b. *Note that relative paths should not be used on OSX.*

A simple profiling aspect

Now that we have some level of comfort with the `javaagent` mechanism, it's time to turn our attention to actually modifying byte code so that we can build a simple profiler. This might seem like a rather daunting task, since few developers have had any reason in the past to get familiar with all of the details of Java byte code or the Java class file format[2] For this article, we'll use the ASM[3] library from the Jonas project because it doesn't require that you necessarily become familiar with every last detail of the class file format.

We'll start out by creating an agent that outputs the class name, method name and the system clock time every time that we enter or leave a method. Here's a simple class we can call to do that.

```
public class Profile {

      public static void start(String className, String methodName) {
            System.out.println(new StringBuilder(className)
                  .append('\t')
                  .append(methodName)
                  .append("\tstart\t")
                  .append(System.currentTimeMillis()));
      }

      public static void end(String className, String methodName) {
            System.out.println(new StringBuilder(className)
                  .append('\t')
                  .append(methodName)
                  .append("\end\t")
                  .append(System.currentTimeMillis()));
      }
}
```

[2] http://java.sun.com/docs/books/vmspec/html/ClassFile.doc.html
[3] http://asm.objectweb.org/

What we want to do is to inject byte code so that this method is called at the beginning of each method.

```
Profile.start("MyClass", "myMethod");
```

But if we're going to inject this into each method, what does the byte code for this look like? It turns out that ASM has a Bytecode Outline plugin for Eclipse that allows you to view the byte code of any class or method. We can use this to translate `Profile.start(..)` into byte code. [For those of you who consider yourself "old school", i.e., "I don't need no stinkin' IDE" there are a couple of other tools that you can use to view the byte code of a class. One is JAD[4] (the JAva Disassembler). However, if you are truly a hard code Java developer you can always use javap[5], the java class file disassembler that ships with the JDK.]

… blah, blah .. screen shot of the byte code ,,,,

Once you know what byte code you need to inject into each method, using ASM to do that is quite easy. *(Not sure I even need to show the example code here since it's pretty simple but at the same time, it's not all that compact)*

Without any space constraints, this is what I'd talk about:

1. *Extending MethodAdapter to inject the byte code.*
2. *Create a ClassAdapter to all integration with our existing 'premain' class.*
3. *Update the manifest*
4. *Run through an example of using these new classes*
5. *Sidebar*
    a. *Using this approach, you can only use static methods.*
    b. *Debugging + line numbers = problems.*

Tracking the call stack

Now we have all of the basic ingredients of a simple aspect oriented profiler. In fact, at this point you can see how easy it would be to build your own functionality by creating your own agents. For example, you could build a logging aspect that would be easy to integrated into existing code.

We are, however, missing two very import pieces of information that we'll need in order to build a useful profiler: thread and call stack information. Call stack information is necessary to distinguish between the total amount of time that a method spends executing verses that amount of time that was spent executing calls to other methods (i.e., gross execution time verses net execution time). In addition, each call stack is associated with a thread, so you need to track thread information as well. Also, in order to track the amount

[4] http://www.kpdus.com/jad.html
[5] http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javap.html

of time from the beginning of a method call to the end of a method call, we'll need to keep track of method start times rather than just dumping them to standard out.

To Do:

1. *Modeling the stack frame, method tracking and computing metrics*
2. *Updating the Profile class to gather the information.*
3. *Adding a shutdown hook to dump the information when the VM exists.*
4. *Sidebar:*
    a. *don't profile yourself.*
    b. *The profiler class can only be "seen" by classes that are loaded by the application classloader while it is possible to inject calls to the Profiler class into classes that are loaded by other classloaders.*
    c. *Different environments, i.e., standalone applications verses web applications, use different classloader hierarchies.*

The groundwork for a low overhead, interactive profiler

At this point we've constructed a very simple Java code profiler. You've also seen how easily aspects can be added to Java code using the Java5™ agent interface. There are, however, a few other things that we'd like to add to our profiler:

1. Interactivity. In order to get really good profiling information for things like web applications, we need to be able turn the profiler on and off while the application is running. This could be accomplished relatively easily by having a TCP socket that would receive commands for turning the profiler on and off.
2. Filtering. We really need to be able to disable profiling for certain classes. When the profiler agent is instrumenting methods, it would be easy to make a check to determine if a class (or method or package) should be instrumented or not.

The purpose of this article is not to go into great detail about how to do these things, but just to lay the groundwork. I've extended the work done here and included the things we didn't go over in an Open Source project called JIP.

JIP – the Java Interactive Profiler

JIP can be downloaded from Sourceforge: http://sourceforge.net/projects/jiprof. If you look at the source code, you will see at lot things that should look very familiar. JIP extends the concepts shown here to include:

- Interactive profiling.
- Filtering by Class or Package.
- Text file output that gives detailed call stack information as well as rolled up summarizations.
- A facility for tracking object allocations.
- The ability to archive profiles.
- An XML output option that makes it easy to write programs that build on JIP's output.
- Performance measurement in addition to code profiling.

<< *not sure I about this:>>*
Even if you don't need interactive profiling or fancy XML output, JIP is a worthwhile profiler simply because you can choose not to profile certain classes. Why is this so important? It's important because small methods are very difficult to accurately profile with any tool. A program with one of these methods that gets called thousands of times is simply impossible to profile in any meaningful or accurate way. That is, unless you can exclude it from being profiled.

<*Summary and conclusion here*>