

JIP – The Java Interactive Profiler

An effective new configurable profiler for Java

Andrew Wilcox
Senior Architect
MentorGen LLC
November 2005

Introduction

This paper 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. Finally, the Java Interactive Profiler is described and how JIP meets these needs is described.

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 your code. There are people who think that a 2% improvement here and a 5% improvement here will add up to a significant 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 were 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 observes 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.

JIP – The Java Interactive Profiler

For the reasons stated above, I developed a new tool called JIP – the Java Interactive Profiler. You can download JIP from its SourceForge homepage: <http://sourceforge.net/projects/jiprof>. JIP is an Open Source, 100% Java solution for code profiling. It uses the `-javaagent` VM option that is new in Java 5™. This option allows the developer to “hook” the classloader and modify byte code when a class is loaded. Basically JIP adds a profiling *aspect* to every method. This allows it to accumulate profiling information. Being Aspect Oriented, JIP is therefore also buzzword compliant! At a high level, JIP is:

- Interactive. JIP can be turned on and off interactively at run time. This allows the developer to make multiple measurements at run time (rather than the single measurement that most profilers give).
- 100% Java. JIP contains no native components and will run on any platform that supports Java 5™.
- Open Source. JIP is licensed under a BSD style license that is arguably one of the most “friendly” open source licenses in use today. You can download JIP for free and even modify the source code if you’d like.
- Lightweight. On average, JIP is 10 to 20 times faster than using `hprof`. An added benefit to JIP is that it can be interactively turned on and off while the target program is running. With profiling turned off, there is almost no overhead associated with using JIP.
- JIP allows you to filter by class or package. Rather than having to wade through a whole bunch of irrelevant data, JIP lets you filter out classes and packages that you don’t want to see in the output.
- Performance timings. JIP is not only a lightweight profiler, but it also tracks the overhead associated with gathering performance data. It factors this overhead out then presenting performance measurements. This makes JIP one of the only profilers that can be used to gather accurate performance timings.

Profiling with JIP

Using JIP to profile code

The first thing that you will need to do is grab the latest release of JIP from Source Forge (<http://sourceforge.net/projects/jiprof>). The release I'll be using is 0.9.2. There are two main directories:

```
/client  
/profile
```

The profile directory contains all of the jars needed for profiling code along with sample properties files that are used to set the initial behavior of the profiler. The client directory contains the files needed to interact with the profiler remotely.

To enable the use of the profiler, you need to use the `-javaagent` switch when the VM is started. The syntax for this is:

```
java -javaagent:%PROFILE-DIR%\profile.jar ...
```

where `%PROFILE-DIR%` is the path to the `profile` directory. **Do not** put this directory in the Extensions loader path and **do not** put the jar files in the application classpath. This might seem counter intuitive, but the java agent will load the specified jar file and the other jars in the `%PROFILE-DIR%` directory using the application classloader. The default behavior is for the profiler to start out being turned on and the remote interface being off (The remote interface allows you to turn JIP on and off at runtime). This will make JIP behave the same way that `hprof` does.

For Web Applications

To profile a web application that is being run inside Tomcat 5.5, you will have to set the `javaagent` option by using the `JAVA_OPTS` environment variable. Under Windows™, you would do this:

```
SET JAVA_OPTS=-javaagent:%PROFILE-DIR%\profile.jar
```

When profiling a web application you probably want to change the default behavior of the profiler. In particular, you probably want the profiler to start out turned off and the remote interface to be turned on. In the `/profile` directory there is an example of a profile properties file for just these settings. It is called `webapp.profile.properties`. Inside it you can see these settings:

```
profiler=off  
remote=on  
port=15599
```

You can tell JIP to use these setting by using the `-Dprofile.properties` VM parameter:

```
SET JAVA_OPTS=-javaagent:%PROFILE-DIR%\profile.jar  
-Dprofile.properties=%PROFILE-DIR%\webapp.profile.properties
```

Using JIP with the default settings (profiler on) is pretty straightforward. JIP starts profiling when your program starts and finishes when the VM terminates. Using JIP interactively, like for a web application, is not difficult at all. Let's say that you have a web application where one of the pages is particularly slow and you want to understand why. You'd start Tomcat with the `JAVA_OPTS` set as described above. In order to get a true measure of performance, you'll want to exercise the page in question a number of times. This avoids measuring the classloading of classes that are used for the page. It also avoids any compilation that the web container would do to support JSPs. I personally will exercise a page at least six times before I run the profiler. This seems to give the JIT enough time to compile all of the parts of the program that are used. But don't take my advice. You can profile the first page hit, and the next and the next until you see the performance level off. Six just seems to be the magic number in my environment. Your mileage may vary. There are a number of batch files in the `/client` directory that allow you to interact with the profiler. These are very straightforward one-line files that are easy to port to Unix. The first thing you needed to do is set the name of the output file if you don't want to use the default. You can do this with:

```
File.bat localhost 15599 c:\tmp\test-profile.txt
```

(Changing the host name, port number and file name as appropriate). Doing this does not start the profiler which you can do it at any time. If you are looking at Tomcat's `stdout`, you should see a one line message that acknowledges that the file name as been set. Next after you've exercised the page to be tested a number of times, you can start the profiler with:

```
Start.bat localhost 15599
```

This doesn't actually start the profiler – there's no need to race from the console window back to the browser to run the test case! This command actually tells the profiler to start profiling the next time code is executed. After you've run your test, you can stop the profiler and dump the output with:

```
Finish.bat localhost 15599
```

Again, there is no need to rush back from the browser to the console to execute this command. Assuming that you are running a private instance of Tomcat that no one else is using, since no code is executing, there is nothing for the profiler to measure.

You can repeat these commands as often as you need to without needing to start or stop the web server. Personally, I almost always start my instance of Tomcat with the java agent set. That way if I'm doing something and decided I want to profile it, I don't even

need to restart Tomcat! When Tomcat is started with the java agent parameter, but with the profiler itself off, you really only notice a slow-down when the Tomcat is starting up (this is because the classes are being rewritten). Other than at start up, it's really no slower than without the profiler.

Using JIP on Third Party products

Since you don't need the source code to profile using JIP, you can actually profile third party products. If you don't have the source code, this is just an academic exercise, but you never know when this ability might come in handy. There is an example of this that ships with the source distribution of JIP. The example is a profile of Ant compiling JIP. You can run this using `example.bat`.

Analysis: using the output of JIP to improve your code.

JIP output explained

The call graph of a JIP output file (see appendix A) contains call graphs by thread. The columns are, from left to right:

- Call count. The number of times that the given method called by the enclosing method.
- Total time. The time, in milliseconds, that was taken up by the execution of this method.
- Net Time. The amount of time that was actually spent executing this method if you factor out the total time taken by calling other (listed) methods. It important to note that the net time is the total time less the sum of the total times for all of the listed "child" methods. The reason that this is important is that some called methods aren't listed and are therefore part of a method's net time. Classes loaded by the bootstrap and extensions classloaders are never listed. If you have chosen to filter out some classes or packages that are loaded by the (web) application classloader, their times will also be reflected in the method's net time.
- Total percent. The total time for this method expresses as a percent of the total time for the given thread.
- Net percent. The net time for this method expressed as a percent of the total time for the given thread.

Because some profiles can produce deep call graphs, the depth of the call graph can be controlled by the **thread-depth** property in the profile properties file. Any positive integer can be used. -1 indicates that there is no limit of the depth of the call graph (this is the default). "compact" can be used to limit the call graph depth by only showing nodes that have a certain minimum *total* time. This minimum total time can be set via **thread.compact.threshold.ms**. The default is 10 milliseconds.

The next (middle) section (see appendix B) is a flattened version of the call graph, ordered by net time. The format and content are almost identical to the bottom of the output for hprof. The number of methods that are output here can be controlled by the profile property **max-method-count**. If not specified, the default is -1 which means unlimited. Here you may also specify "compact" which will only display methods that

have a certain minimum *net* time. The default for this is 10 milliseconds, but can be changed via **method.compact.threshold.ms**.

The bottom section (see appendix C) looks like the middle section but there is an important difference. In the top section (the call graph), you may notice that a given method may appear more than once if it is called from different places in the code. The middle section is just a flattened version of the top section, so a method may appear more than once in that section as well. The bottom section is a summarized list by method. This means that a method that appears three times in the middle section will appear only once in the bottom section. The count, net and percent times are the sum of the information from the previous section.

How to optimize

Even with all of this information, it's easy to get side tracked and start to optimize things that don't really matter. Keep this simple rule in mind with profiling:

- Only optimize things that are truly expensive

The two top sections are really there just to give context to the bottom section. The bottom section gives a list of method calls ordered from the most time consuming to the least time consuming. Look at the third column – percent. This brings us to our next profiling rule:

- Only pay attention to methods that use 10% or more of the total time.

If a method cost 2%, you can optimize it until it goes to zero but you're really doing next to nothing to help the overall performance of the application. After you have isolated methods that need to be looked at, use these two simple rules:

- If the call count is high, look into ways to lower the number of calls that are made.
- If the call count is low, look at the method to see if there are any local optimizations that can be made.

If all the calls have a net time of 5% or less, look at optimizing the main algorithm. This is also a good time to look OS level metrics like disk and network I/O.

When not to trust the Profiler

The profiler is a valuable tool, but there are situations when the information it gives you is suspect. In particular, methods that execute quickly are difficult to profile because their total execution time may fall below that threshold of measurability. Look for methods that have a high call count but a low net time. When I profile, it's not uncommon for me to run the profiler three times to try to get a better understanding of the average performance of a piece of code. If you do this and notice that the time for a particular method varies widely than you would expect, and the method has a relatively high call

count, it's good to be skeptical about the validity of the measurement. Don't spend too much time trying to optimize methods that are like this.

Hacking JIP: Insights to how JIP was built

JIP is composed of four parts that can be broadly described as:

- Instrumentation
- Runtime profiler
- Output generator
- Client

Instrumentation

Instrumentation is the part of JIP that adds the profiling “aspect” to classes as they are loaded by the classloader. It uses Java 5™'s `-javaagent` command line option to do this. This option is not very deeply documented and so there were a number of things that I had to learn by trial and error¹. Here's a rundown of many of the things I had to figure out:

- In the manifest file, you list a `Boot-Class-Path`. From doing command line Java for so long, I kept thinking that those jars must need to be either in the classpath or loaded by the extensions classloader. Neither of these are the case. The jar files simply need to be in the same directory as the jar file specified by the `javaagent` option. See R. J. Lorimer's article for more important information about what needs to be in the manifest file.
- JIP uses the ASM libraries from the Jonas project (asm.objectweb.org). This was really helpful, but since I hadn't done any byte code manipulation there was a little bit of a learning curve. Not being a very patient person probably didn't help either. One important thing I did (eventually) learn is that you can't use ASM to add local variables to a method. This meant that all profiler calls had to be static. This wasn't a problem, but it took me a while to figure that out.
- Class visibility. This is a **BIGGIE** and took me longer than I care to mention to figure out. The Instrumentation, Runtime Profiler and Output Generator are all part of `profile.jar` that is loaded by the java agent mechanism. For some reason I was under the mistaken impression that these classes were loaded by either the bootstrap classloader or the extensions classloader. In reality (and this is very plainly spelled out in Sun's docs), these classes are loaded by the application classloader. Why is this important? The instrumentation piece “sees” classes that are loaded by the extensions classloader as well as classes that are loaded by the application classloader. The issue is that it was adding static calls to the `Profile` class that, since the `Profile` class was loaded by the application classloader, weren't “visible” to classes that were loaded with the extensions classloader. This led to `ClassNotFoundException` exceptions occurring at runtime. This really drove me nuts until I figured this out. The next trick is to figure out, when a class is classloaded, which classloader is loading it. It turns out that the static method `ClassLoader.getSystemClassLoader()` returns the application classloader which enables you to tell which classloader a class that is being instrumented is being loaded by (the loading classloader is a parameter to the `transform()` method from

which the classes are instrumented). The next problem is that things are not nearly that simple for web applications. There is a deeper hierarchy of classloaders and the names of these classloader are not in the Servlet spec (the bootstrap classloader, extensions classloader and application classloader hierarchy are defined as part of the VM spec). The way I got around this is a bit of an application specific kludge. I look for the VM parameter “catalina.home” which is used by Tomcat, to see if we’re running in a web app. If we are, I know which classloader Tomcat 5.5 uses to load classes for each webapp, so I test for that. This works well since developers don’t need to do anything to configure for Tomcat verses standalone applications. The downside is that this won’t work in other environments.

Runtime profiler

- The algorithm for gathering profile data is very simple. It is simple because it is based on the assumption that the order in which methods are called mirrors the call stack exactly. This simplifying assumption makes JIP lightweight. However, there are a couple of cases where this assumption breaks down. The first should be obvious. When an exception is thrown the flow of control is interrupted. The other situation that breaks this assumption might not be as obvious. Static initializers are not invoked as part of the flow of control; they are invoked by the classloader. This really gets you in trouble when profiling a standalone application where the “main” class has a static initializer, like Apache Ant. This means that you have to factor out static initializers.

Client

- The client interaction with the profiler at runtime is very simple. It’s just a socket connection. The Profiler has a certain set of commands that it understands and can be used to manipulate the profiler at runtime.

Future Direction

Appendix A – The call graph section of the profiler output

```

+-----+
| File: ant-profile.txt
| Date: 2005.12.02 07:08:19 AM
+-----+

+-----+
| Thread depth limit: Compact
+-----+

+-----+
| Thread: 1
+-----+

+-----+
| Time          Percent
+-----+
| Count  Total    Net    Total    Net    Location
+-----+
| 1      2909.0    4.4    100.0    0.2    +--Main:main (org.apache.tools.ant)
| 1      2904.6    6.6     99.8    0.2    | +--Main:start (org.apache.tool
| 1      2874.6    31.0    98.8    1.1    | | +--Main:runBuild (org.apache.tool
| 1         5.1    0.1     0.2    0.2    | | | +--Main:addBuildListeners
| 1         5.0    4.7     0.2    0.2    | | | | +--Main:createLogger (org.ap
| 1         2.0    2.0     0.1    0.1    | | | | +--Main:getAntVersion (org.ap
| 1       380.4    0.0    13.1    0.1    | | | | +--ProjectHelper:configureProje
| 1       376.6    4.0    12.9    0.1    | | | | | +--ProjectHelperImpl:parse
| 1       237.6    0.1     8.2    0.1    | | | | | +--JAXPUtils:getParser
+-----+-----+-----+-----+-----+
| 1       237.6    0.0     8.2    0.1    | | | | | +--JAXPUtils:newSAXParser
| 1       201.2    0.5     6.9    0.6    | | | | | | +--SAXParserFactoryImpl
| 1       200.7    17.2    6.9    0.6    | | | | | | | +--SAXParserImpl:<ini
| 1       183.5    0.0     6.3    0.6    | | | | | | | | +--SAXParser:<init>
| 1       183.4    2.6     6.3    0.6    | | | | | | | | | +--SAXParser:<ini
| 1       179.2    0.0     6.2    0.6    | | | | | | | | | | +--ObjectFactor
| 1       179.2    2.1     6.2    0.6    | | | | | | | | | | | +--ObjectFact
| 1       173.9    0.2     6.0    0.6    | | | | | | | | | | | | +--ObjectFa
| 1       172.7    37.4    5.9    1.3    | | | | | | | | | | | | | +--Object
| 1       126.7    0.0     4.4    0.6    | | | | | | | | | | | | | | +--XML1
| 1       126.7    1.5     4.4    0.6    | | | | | | | | | | | | | | | +--XM
| 1       125.2    0.0     4.3    0.6    | | | | | | | | | | | | | | | | +--
| 1       124.2    0.0     4.3    0.6    | | | | | | | | | | | | | | | | | +

```


Appendix C – The most expensive methods

1	2.1	0.1	org.apache.xerces.util.ObjectFactory:createObject
1	2.0	0.1	org.apache.tools.ant.Main:getAntVersion
162	2.0	0.1	org.apache.tools.ant.Project:fireMessageLoggedEvent

Summarized			

	Net		
Count	Time	Pct	Location
=====	=====	=====	=====
1	1373.5	47.2	org.apache.tools.ant.taskdefs.compilers.Javac13:execute
1	612.3	21.0	org.apache.tools.ant.Project:init
48	83.0	2.9	org.apache.tools.ant.taskdefs.Zip:zipFile
9	51.0	1.8	org.apache.tools.ant.util.FileUtils:copyFile
1	37.4	1.3	org.apache.xerces.util.ObjectFactory:newInstance
5	36.6	1.3	org.apache.tools.ant.IntrospectionHelper:<init>
1	36.3	1.2	org.apache.tools.ant.util.JAXPUtils:newParserFactory
1	34.7	1.2	org.apache.xerces.impl.dtd.XMLDTDValidator:<init>
1	34.1	1.2	org.apache.xerces.impl.XMLDocumentScannerImpl\$XMLDeclDi
484	33.4	1.1	org.apache.tools.ant.types.selectors.SelectorUtils:match
1	31.0	1.1	org.apache.tools.ant.Main:runBuild
6	24.1	0.8	org.apache.tools.ant.taskdefs.Delete:removeDir
1	21.4	0.7	org.apache.tools.ant.Diagnostics:validateVersion
3	20.8	0.7	org.apache.xerces.impl.XML11EntityScanner:skipSpaces
1	20.2	0.7	org.apache.xerces.parsers.DTDConfiguration:createEntity
1	17.3	0.6	org.apache.tools.ant.Project:<init>
1	17.2	0.6	org.apache.xerces.jaxp.SAXParserImpl:<init>
2	12.5	0.4	org.apache.tools.ant.taskdefs.Zip:execute
1	10.3	0.4	org.apache.xerces.parsers.IntegratedParserConfiguration
58	10.2	0.3	org.apache.tools.ant.util.FileUtils:resolveFile
1	9.1	0.3	org.apache.xerces.impl.XMLDocumentScannerImpl:<init>
1	8.6	0.3	org.apache.xerces.util.ObjectFactory:findProviderClass
9	8.2	0.3	org.apache.tools.ant.DefaultLogger:printMessage
1	7.4	0.3	org.apache.tools.ant.util.SourceFileScanner:restrict
24	7.3	0.3	org.apache.tools.zip.ZipOutputStream:getBytes

ⁱ See “Instrumentation: Modify Applications with Java 5 Class File Transformations” by R. J. Lorimer (<http://www.javalobby.org/java/forums/t19309.html>) for some really good information on this topic.