

JINSPIRED Developer

Software Metering, Metric Monitoring and Event Signaling for Java Developers

[Builds](#) [Usage](#) [Docs](#) [Repo](#) [API](#) [Install](#) [Config](#) [Console](#) [Site](#)

If you're not metering you're not trying hard enough to be the best – Part 2 of 3

by Architect on August 22, 2011

In the [first part](#) of this 3 part series on why intelligent activity metering is the only viable [application performance management](#) approach, a number of Java code profiling solutions were tested using a focused and somewhat contrived benchmark to determine the best case (lowest) unit cost in the measurement (*profiling*) of an instrumented method call. In closing [part 1](#) it was put forward that there was only so much efficiency that could be squeezed out of measurement code and that it was never going to be enough to handle real-world usage, especially in low latency environments with high throughput requirements and large code coverage, unless measurement was not performed and instrumentation magically disabled (*even removed*) based on runtime intelligence (@see [LAM](#)) gathered during the actual running of the application.

Benchmarks

In this part I've have used two standard benchmarks from the [SPECjvm2008](#) benchmark suite for comparison purposes. Here is a how the benchmark suite is described in the supporting documentation.

"SPECjvm2008 is a benchmark suite, containing several real life applications and some benchmarks focusing on core java functionality. The main purpose of SPECjvm2008 is to measure the performance of a Java Runtime Environment (JRE). It also measures the performance of the operating system and hardware in the context of executing the JRE. It focuses on the performance of the JRE executing a single application; it reflects the performance of the hardware processor and memory subsystem, but has low dependence on file I/O and includes no network I/O across machines. The SPECjvm2008 workload mimics a variety of common general purpose application computations. These characteristics reflect the goal that this will benchmark be applicable to measuring basic Java performance on a wide variety of both client and server systems running Java."

Benchmarks designed to measure the performance of the Java Runtime Environment are the worst case examples for any Java code profiling or application performance management (APM) solution as they usually exhibit the high frequency execution of many fine grain methods (*getters, setters,...*) and have very little network traffic which many application performance solutions depend on to mask their relatively (*to the method level*) high instrumentation, measurement and collection cost. *They're the things that nightmares are made out of.*

I selected two benchmarks from the suite – *sun flow (cpu-bound)* and *derby (gc & db bound)*. Here is how both are described in the [SPECjvm2008](#) documentation.

"This [sunflow] benchmark tests graphics visualization using an open source, internally multi-threaded global illumination rendering system. The sunflow library is threaded internally, i.e. it's possible to run several bundles of dependent threads to render an image."

"This [derby] benchmark uses an open-source database written in pure Java. It is synthesized with business logic to stress the BigDecimal library. It is a direct replacement to the SPECjvm98 db benchmark but is more capable and represents as close to a "real" application. The focus of this benchmark is on BigDecimal computations (based on telco benchmark) and database logic, especially, on locks behavior."

Solutions

I ran the benchmarks with 3 different Java method level profiling solutions: JXInsight/OpenCore, DTrace and NetBeans Profiler.

The following configuration was used with the JXInsight/OpenCore benchmark test runs:

```
jxinsight.server.probes.console.enabled=true
jxinsight.server.probes.meter.clock.time.include=false
jxinsight.server.probes.meter.jvm.clock.time.include=true
jxinsight.server.probes.meter.jvm.clock.time.name=clock.time
jxinsight.server.probes.aggregates.enabled=false
```

Instrumentation of all classes other than those on the *bootclasspath* was performed using the following VM arguments:

```
-agentpath:/OpenCore/bin/osx-32/libjxinsight.jnilib=prod -javaagent:/OpenCore/bundle/java/default/opencore-ext-aj-
javaagent.jar
```

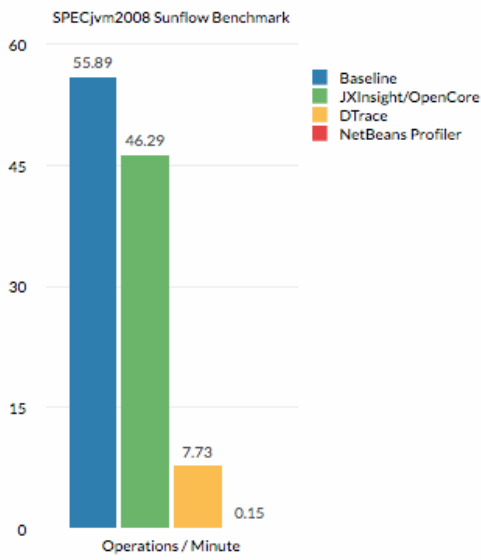
For the DTrace benchmark test runs all that was enabled was the native JVMTI integration via the VM argument `-XX:+ExtendedDTraceProbes`. No DTrace script was used which means this is the absolute best (*and completely worthless*) case for DTrace in offline (*non-observant*) mode for production monitoring of Java applications and runtimes. For this reason it can't be compared with the NetBeans Profiler benchmark results but is included here to show the overhead of instrumentation without measurement and refute claims that DTrace has near zero overhead.

For the NetBeans Profiler benchmark test runs I used the VM arguments generated within the NetBeans IDE for local and direct integration excluding all *bootclasspath* classes.

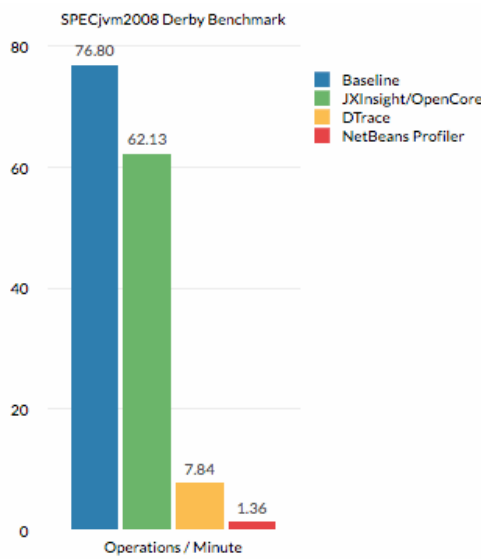
Results

Due to some issues in running the benchmarks with the latest Java 6 update I was forced to execute all benchmarks using only a single concurrent benchmark thread which can be done at the command line of the [SPECjvm2008](#) script using the `-bt` flag.

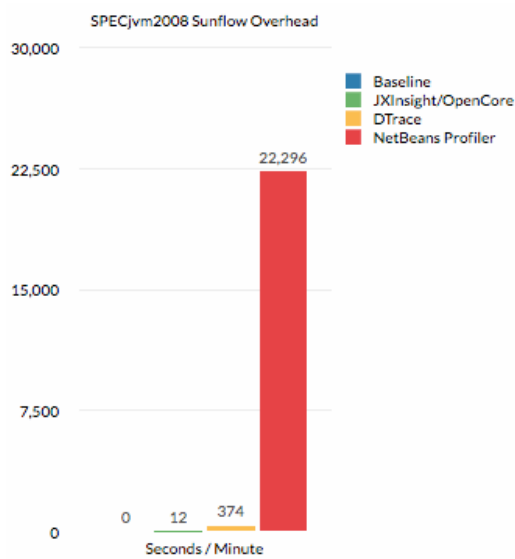
Below are the best *sunflow* benchmark test run results, *operations per minute*, for each of the solutions alongside a baseline (*no solution*) run. With OpenCore the throughput dropped down to 83% which might seem not very good until you consider that DTrace which did effectively nothing had a throughput of just 13% and NetBeans 0.27%.



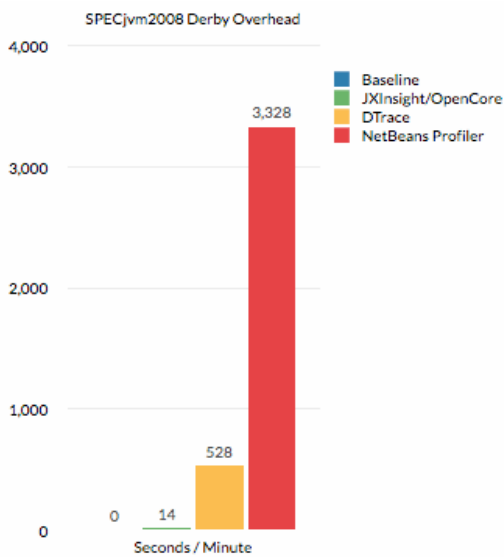
The following chart presents the best derby benchmark test run results, *operations per minute*, for each of the solutions compared with a baseline (*no solution*) run. Compared with the baseline throughput OpenCore is 81%, DTrace 10% and NetBeans <2%.



A much better way of looking at the above results is to determine the additional time, in seconds, that would have been spent in executing the same level of activity performed by the baseline within a minute interval. In the *sunflow* chart below OpenCore would have taken an additional 12 seconds for each 60 seconds of baseline work performed. DTrace would have incurred an additional time lag of 374 seconds (*over 6 minutes*) and NetBeans 22,296 seconds (*over 6 hours*).



Here is the overhead comparison chart for the *derby* benchmark results. OpenCore 14 seconds. DTrace close to 9 minutes. NetBeans close to 1 hour with 55 minutes.



To put the above results into perspective we were able to estimate with a reasonable level of accuracy the number of calls made per *sunflow* operation per minute. This was 0.6 billion. This gives a total of 34 billion calls made in achieving the *sunflow* baseline of 55.89 operations per minute. Over 6 minutes the benchmark would have executed approximately 200 billion calls to methods in `org.sunflow.*`. With the calculated overhead of 12s per minute for OpenCore this gives a unit cost of 0.35 ns (12 / 34) compared to a NetBeans cost of approximately 655 ns making that 20% drop in throughput actually pretty impressive.

Relevance

OpenCore was able to achieve such amazing results by being *strategy* based. Whilst all methods in the `org.sunflow.*` and `spec.*` packages were instrumented only a few were still being metered by the end of the benchmark due to the dynamic *hotspot* classification and disablement of probes by OpenCore's metering engine based on actual real-time profiles – *no guesswork and full code coverage*.

The *hotspot* classification and disablement is shown in the metering model exported on completion of one of benchmark test runs. Instead of metering 28 billion calls per minute only 15 million were, adding approximately 3s-4s of overhead per minute, with the remaining overhead (> 8s) attributed to the short circuiting of metering within the instrumented methods. The total metering count represented less than 0.05% of the total method innovation count but the most important 0.05%.

Metering Table				
Context	Probe	Meter	Count	
1515	org.sunflow.core.gi.InstantGI.getIrradiance	clock.time	9,781,701	
1515	spec.harness.Launch.runOneBenchmark	clock.time	2	
1515	spec.harness.ProgramRunner.runIteration	clock.time	3	
1515	org.sunflow.core.renderer.BucketRenderer.render	clock.time	277	
1515	org.sunflow.core.shader.DiffuseShader.getRadiance	clock.time	9,781,701	
1515	org.sunflow.core.LightServer.getRadiance	clock.time	9,244,044	
1515	org.sunflow.core.LightServer\$1.run	clock.time	1,108	
1515	org.sunflow.core.renderer.BucketRenderer.refineSamples	clock.time	*10,719,900	
1515	org.sunflow.core.renderer.BucketRenderer.computeSubPixel	clock.time	9,244,044	
1515	org.sunflow.core.ShadingState.diffuse	clock.time	9,781,701	
1515	org.sunflow.core.LightServer.calculatePhotons	clock.time	554	
1515	org.sunflow.core.accel.KDTree.build	clock.time	554	
1515	org.sunflow.core.LightServer.getIrradiance	clock.time	9,781,701	
1515	org.sunflow.core.Scene.getRadiance	clock.time	9,244,044	
1515	org.sunflow.core.ShadingState.getIrradiance	clock.time	9,781,701	
1515	org.sunflow.core.renderer.BucketRenderer.renderBucket	clock.time	4,432	
1515	org.sunflow.Benchmark.imageUpdate	clock.time	4,432	
1515	org.sunflow.core.shader.MirrorShader.getRadiance	clock.time	662,861	
1515	org.sunflow.core.Geometry.intersect	clock.time	12,143	
1515	org.sunflow.core.accel.BoundingIntervalHierarchy.intersect	clock.time	17,357	
1515	org.sunflow.core.light.TriangleMeshLight\$TriangleLight.getSamples	clock.time	5,018	
1515	org.jfree.chart.ChartFactory.createXYLineChart	clock.time	2	
1515	org.sunflow.core.tessellatable.BezierMesh.tessellate	clock.time	277	
1515	org.sunflow.core.LightServer.initLightSamples	clock.time	7,004	
1515	org.jfree.text.G2TextMeasurer.getStringWidth	clock.time	28	
1515	org.sunflow.core.shader.ClassShader.getRadiance	clock.time	34,011	

Revisiting

The ability to adapt does not end with selective metering (*measurement*) and its collection. OpenCore has an unique ability to use the metering profile of a run to generate a new class or method level instrumentation set.

For class level refinement you can use the `-generate-aj-filters` console command to generate a new `jxinsight.aspectj.filter.config` based on the hotspot classification and probe disablement/enablement within a metering model.

```
java -jar opencore-console.jar -generate-aj-filters ${snapshot} > jxinsight.aspectj.filters.config
```

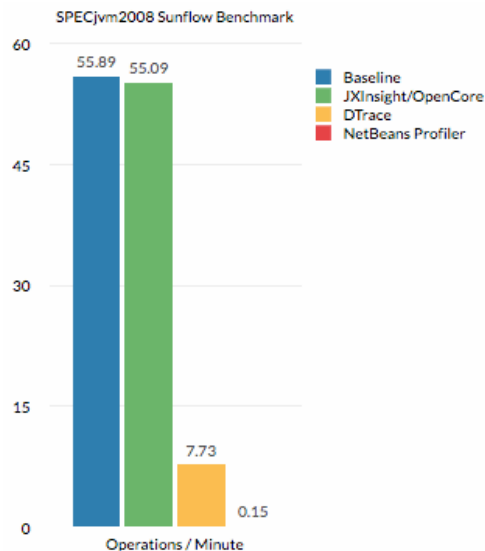
Alternatively you can use a method level approach by generating a completely new agent bundle using the `-generate-aj-bundle` console command.

```
java -jar opencore-console.jar -generate-aj-bundle ${snapshot} ${bundle}
```

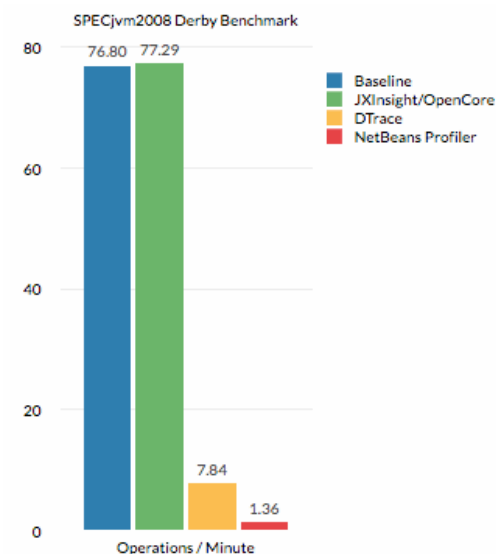
When run the command will analyse the metering model within the snapshot specified, determine a refined probe instrumentation plan, then load the specified agent bundle updating its probes related aop.xml with a more selective set of methods (*pointcuts*) and weaved classes and finally saving the revised agent bundle, `opencore-ext-aj-javaagent.jar`, to the current working directory.

Note: The ability to automate the generation of an instrumentation plan is a unique feature of our solution but one that does require a good performance test/load environment to be put in place and maintained. Unfortunately many application performance test tools are not able to generate workload that is truly representative of production and cause the same degree and level of resource consumption on backends. Sadly most application performance test tools don't get enough performance engineering attention themselves and exhibit significant scaling problems, variability and inaccurate reporting. Why? I think they are simply not trying hard enough to be the best or use the best measurement technology.

Here is the revised result for OpenCore using a new method level agent bundle generated from the initial run. You can have your cake and eat it but only as long as put trust in the runtime and tooling and don't attempt to select what should be instrumented which is the case for all other application performance management solutions.



With the *derby* benchmark re-run OpenCore achieved a higher operations per minute rate than the baseline – an explanation of which is for another day and article.



In part 3 we will look at some commercial application performance management solutions based on legacy tracing & call stack sampling technologies demonstrating the marketing & benchmarking tricks employed to mask much higher overhead levels and highlighting how limited the application of such solutions are outside of poorly architected & developed high latency applications.

Software

[JINSPIRED OpenCore 6.2.6](#)

```
java version "1.6.0_26"  
Java(TM) SE Runtime Environment (build 1.6.0_26-b03-384-10M3425)  
Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02-384, mixed mode)
```

Hardware

```
Model Name: iMac  
Model Identifier: iMac11,1  
Processor Name: Intel Core i7  
Processor Speed: 2.8 GHz
```

Number Of Processors: 1
Total Number Of Cores: 4
L2 Cache (per core): 256 KB
L3 Cache: 8 MB
Memory: 8 GB
Processor Interconnect Speed: 4.8 GT/s

FROM: PROBES
