If you're not metering you're not trying hard enough to be the best – Part 1 of 3

by Architect on August 5, 2011

The title of this blog is a statement that a number of our customers have paraphrased in some form or another when the topic of product comparison comes up in discussions with various teams in development, test and operations. We take the performance of our measurements runtimes very seriously – *we practice what we preach*. There are a number of reasons for this including a significant % of our customer base being financial institutions with low latency requirements in which any additional time (*overhead*) represents a potential loss. But also because we respect important characteristics of information our runtimes provide including accuracy, relevance, immediacy, accessibility, coverage, precision and resolution – a number of which are directly impacted by measurement cost which is the subject of this first in a series of blog entries on information quality in the context of performance measurement (*profiling*).

A Note On Premature Performance Optimization

There are a number of developers who view profiling applied early in the development lifecycle as premature (optimization) and the root of all evil. But they fail too see that profiling is very much like any feedback loop process with 3 distinct steps: observation, judgement and reaction. Optimization is a reaction to an analysis made on performance information collected by a measurement runtime. It is this last stage which could in some cases be classified as premature. From our experience in applying software performance engineering from dev-through-to-production or production-back-into-dev (when we are called in to fix things far too late) we have recognized the value that is offered in collecting (observing) and analysing (judgement) performance profiles as an application grows, matures and is maintained. The journey is just as important as arriving at the final destination and if you have being mindful during this you will be in good stead when you arrive and when you discover that in the meantime things have changed from when you began. Software performance engineering is very much a process of knowledge acquisition of the execution behavior and resource consumption profile of your software. It is important to know how you arrived at the end point and to look back at the milestones which now impact the end result. Action does not always need to be taken until we are sure it does represent a risk to achieving our goals. We can only manage this risk if we know were we currently stand and how we arrived at this situation. It is for these reasons we offer a free development edition license to all of our customers ensuring that when our runtime uncovers important information during pre-production and/or production they are much better equipped and informed to make the right targeted corrective action. Alan Kay is quoted as saying "The biggest problem we have as human beings is that we confuse our beliefs with reality". Profiling should be viewed more so as an exercise in software execution analytics rather than solely optimization allowing us to see beyond the flat plane of source code in our programming editors.

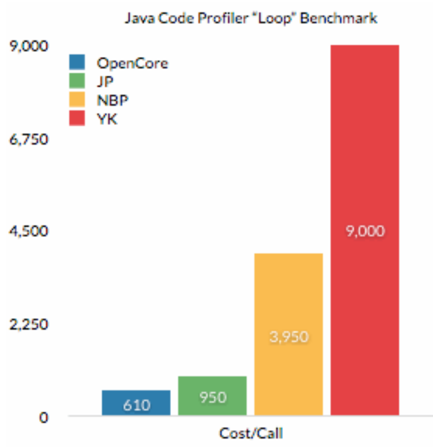
One of the first steps in accessing the quality of any information produced by a performance measurement solution is to determine the information acquisition cost because unfortunately this is the very thing that the solution is trying to measure. To do this we need a controlled experiment that enables us to perform and observe the operation that is of interest and *exclusively* if possible. Here is a snippet of code that we use internally to determine this across solutions.

```
public void run() {

  long start = nanoTime();

  int count = RUN_COUNT;

  for (int i = count; i > 0; i--)
    iterate();

  long end = nanoTime();

  out.println("cost=" + (end-start) / (double)(count));

}

private static void iterate() { outer(); }

private static void outer() { inner(); }

private static void inner() {}
```

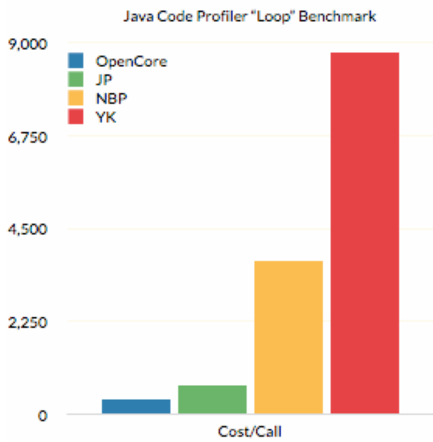
Here is a chart comparing the average nanosecond *overhead* " `cost` " printed by the above code for the 4 most popular Java code profiling solutions including JXInsight/OpenCore. OpenCore has the lowest measurement overhead with a method unit cost of `203ns - 610ns/3`.

***Note**: For the OpenCore test run I disabled the intelligent metering hotspot strategy using* `jxinsight.server.probes.strategy.enabled=false` *ensuring all method invocations were metered.*

***Note**: The YK profiler did not instrument or measure the* `inner()` *method so the cost is understated by 33%.*



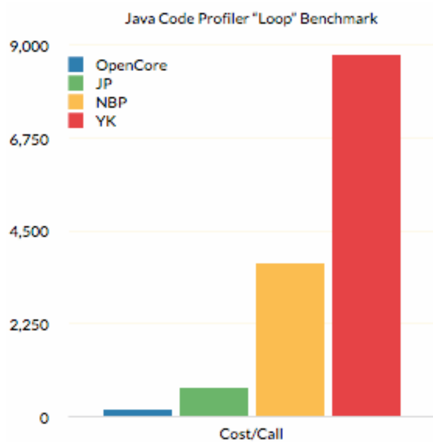
In the next chart I have subtracted away the cost in accessing the wall clock 6 times in executing `iterate()` to get a better picture of the efficiency of each solution. OpenCore is 2 times more efficient than the nearest solution and 20 times more efficient than the least efficient solution.



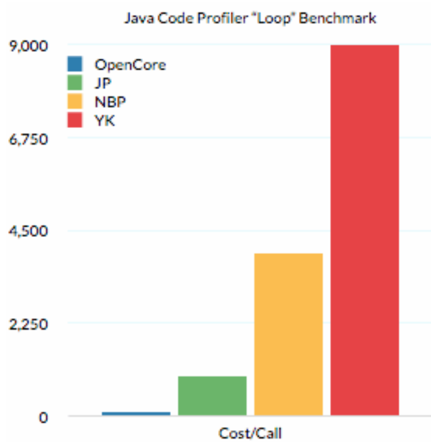
OpenCore does much more measurement (*metering*) aggregation out of the box than other solutions which is made immediately accessible from within the benchmark runtime itself via the Open API. Since this is generally not needed during investigative profiling I have performed another test with the following system properties:

```
jxinsight.server.probes.aggregates.enabled=false
jxinsight.server.probes.global.enabled=false
```

This configuration change reduces the adjusted method unit cost for OpenCore down to `53ns`. OpenCore is now more than 4 times more efficient than its nearest rival and 55 times more efficient than its furthest rival.

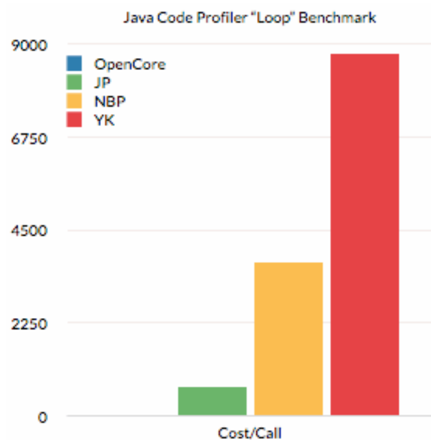


If all we want is to count the execution of the methods the overhead can be further reduced to less than `40ns` using the `callcount` probes provider.



Up to now what we have effectively compared across products is the quality of the engineering involved in improving the efficiency of pretty much boilerplate measurement code. But this tuning will only get you so far. The much bigger cost savings are to be found in design and invention driving the engineering of a solution. This is were OpenCore's metering technology shines alone in its unique adaptive runtime approach to measurement that is driven by an awareness of actual cost, behavioral context and its relevance.

Looking at the above code we would expect an intelligent measurement solution to stop measuring the `iterate()`, `outer()` and `inner()` methods which have no runtime effect whatsoever and which would more than likely be removed by the runtime compiler if it were not for the instrumentation code dynamically weaved into the code by the profiling solution. This is exactly what happens when OpenCore's strategy probes provider is not disabled dropping the method unit cost for OpenCore down to `0ns`.



So how material is this to the quality of profiling information captured for a real world application? Here are a few questions to ask yourself in assessing this.

- Do you need to have the highest level of *confidence* in reporting your findings to team members and other teams involved in a sensitive and time critical problem diagnosis?
- Do you need to have actual measurements from a live production environment framing your software performance engineering efforts during development and test?
- Are the problems you encounter exclusively low hang fruit (*very poor code quality, high latency remote resource access*) that any excessive measurement overhead would unlikely alter your findings?
- Is it important for you and others in your organization to be able to use the same measurement runtime and tooling irrespective of the latency bound of an application and its execution environment? *Do you have to measure high latency business applications as well as a low latency trading, messaging or gaming platform?*
- Can you afford (*or risk*) limiting the instrumentation coverage as a way to reduce the cost overhead? *Do you already have sufficient information (pulled from a hat) that would allow you to restrict instrumentation to the hotspots you are trying to find in the first place?*
- Can you be sure that what you deem as an acceptable measurement cost is unlikely to have a much bigger system wide impact when applied to a highly concurrent and contended/congested code path?
- Do you think it much more efficient and practical for the measurement runtime to find the hotspots and behavior that is of importance driven by policies you define rather than you wading through lots of data of questionable quality and which offers more questions than answers? *Do you really want to repeat this for any problem diagnosis you are involved in?*
- What is the ratio of hotspot to non-hotspots in your instrumented applications? *From our experience hotspots represent a very small (single digit) % of the total instrumented and measured (metered) codebase of any application. Why pay (incur costs) for something you don't need – continuously?*

In part 2 I am going to touch on the importance of *relevance* and to assess the impact of the above unit cost profiles against various application (*response time*) latency bands along with a much more realistic experiment using an industry standard benchmark. In the meantime please check-out this article on *Intelligent Activity Metering (IAM).*

Update 22.8.11 Part 2

Software
JINSPIRED OpenCore 6.2.2

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: BENCHMARKS, LOW LATENCY, PROBES