

# 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 3 of 3

by Architect on September 13, 2011

In the final part of a 3 part series on why [intelligent activity metering](#) is the only viable [application performance management](#) the focus now shifts from low level code profilers benchmarked in [part 1](#) and [part 2](#) to application performance management monitoring solutions which target pre-production and production environments. I've chosen to pick one of the newer products which claims to have a much lower overhead than the current or previous crop of APM solutions.

### Low Overhead Myth

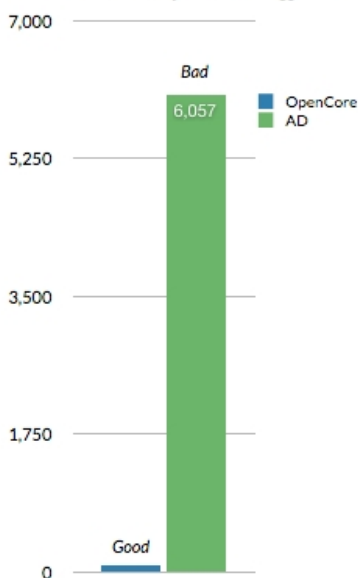
An unfortunate misconception regarding application performance management solutions is that because they target pre-production and production environments they must be much more efficient than a typical low level code profiler. In fact from our investigations this could not be further from the truth as code profilers have not got the luxuries afforded to application performance management solutions such as limited code instrumentation coverage.

The following chart compares the average instrumentation and measurement cost (in nanoseconds) for a single method call executed more than billion times. The unit cost for AD is prohibitively expensive close to 75 times slower than JXInsight/OpenCore with it's *hotpot* strategy disabled (which we never advise doing). Worse still is that AD by default only samples each 100 transaction (method) execution so the unit cost for AD is greatly understated.

AD is 3 times slower the slowest commercial profiler we have benchmarked.

Application performance management solutions are not themselves anymore efficient they simply don't do so much.

Method Level Measurement Adj Overhead - Bigger is Worse



### Contextual (Trace) Profiling

The primary differentiator (though this is changing) between a code profiler and an application performance monitoring solution is in the collection of context associated with the code execution. This is evident in many of the case studies published by application performance management vendors including AD that highlighting how customers solved performance issues that plagued production by simply capturing the SQL statements executed by application code.

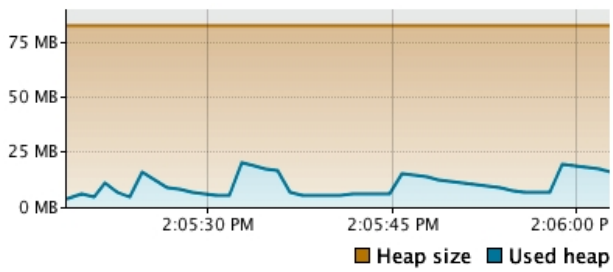
Unfortunately contextual tracking and collection is expensive both in terms of execution time and memory footprint so this generally only happens at coarse grain execution points such as at entry point into a Java Servlet request processing and the exit point of a JDBC query execution. Generally anything in between is simply ignored, lost or worse reported inaccurately using call stack sampling.

I created a small JDBC benchmark, listed [below](#), to compare the cost of contextual metering using the recently [announced](#) JXInsight/OpenCore JDBC-to-SQL probes extension against the top level transaction profiling offered by AD.

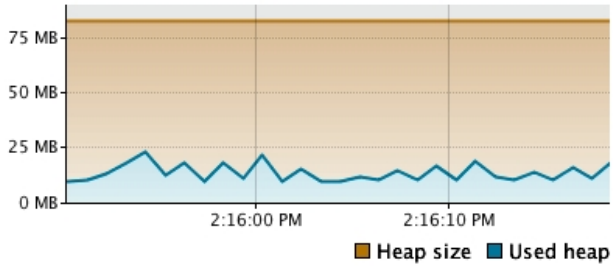
**Note:** JXInsight 2.0 pioneered true JDBC transaction trace path profiling in 2000-2001. Since then we have for the most part abandoned tracing for activity based metering as a much more viable alternative for large scale low latency production monitoring and management.

**Note:** AD unlike OpenCore needs to have a transaction entry point to be manually setup in the AD web console to define a single POJO method representing this point. I used the `iterate()` method in the `jdbc.Benchmark$Runner` class listed below to do so.

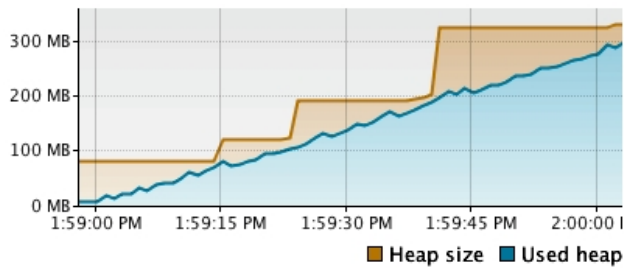
Here is the heap stats collected using an independent JMX based monitoring tool during a non-instrumented test run of the JDBC benchmark and without any specific `-Xmx` setting.



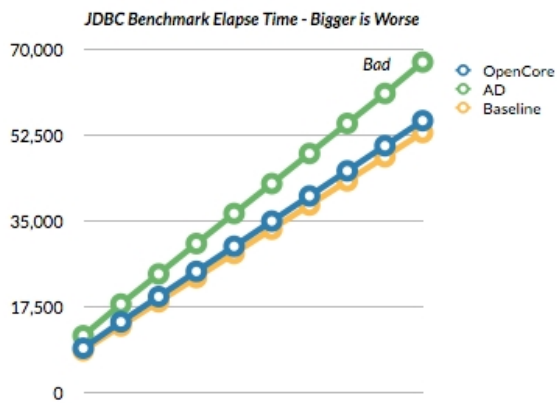
Here is the same chart for the JXInsight/OpenCore test run. There are some subtle differences which can be attributed to the tracking of state related to SQL (including parsing, normalization, and cleaning) across instances of `java.sql.Connection`, `java.sql.Statement` and `java.sql.ResultSet` created and called during the benchmark.



Surprisingly AD crashed repeatedly with `OutOfMemory` exceptions each time we tried to run the JDBC benchmark even when doubly the maximum memory heap. Eventually we managed to get it to complete the benchmark with a setting of `-Xmx356M`.

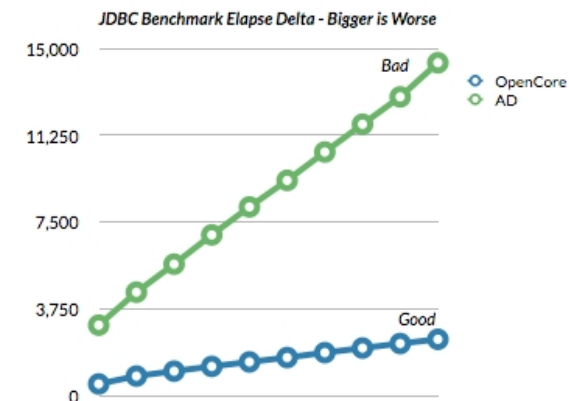


Below is a comparison of the cumulative elapse time of each benchmark test which performs 10 consecutive runs of 10,000 iterations with each performing 7 SQL queries.

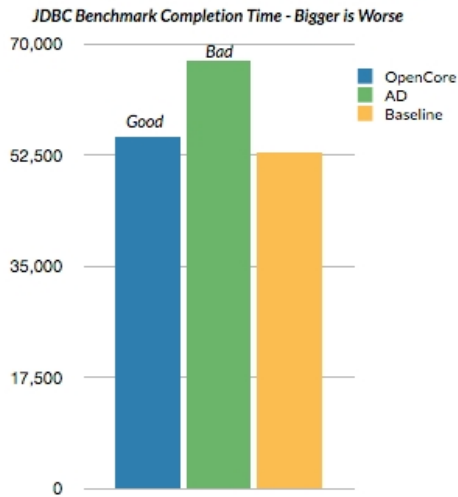


Here is the same data this time only charting the difference in timing from the non-instrumented baseline benchmark run.

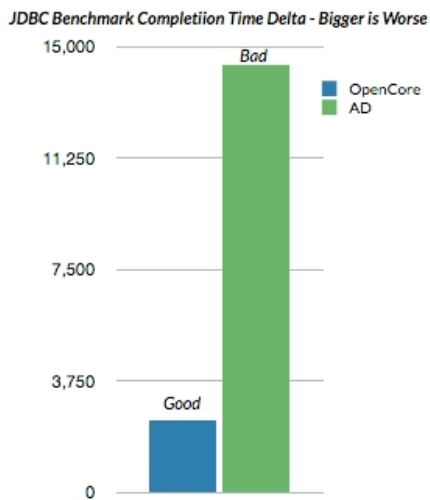
Note: AD unlike OpenCore did not collect any measurements related to `java.sql.Connection` and `java.sql.ResultSet` invocations so the difference could be far greater.



The following chart compares the best completion time of each of the test runs. AD had a significant runtime impact which is a little bewildering considering this is a high latency database bound benchmark.



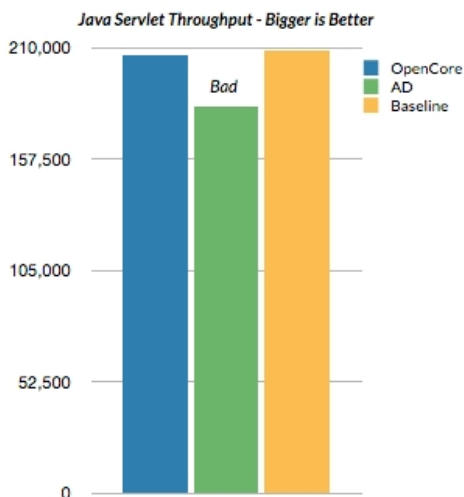
The following chart provides a much truer comparison of the cost efficiency of each solution in subtracting the baseline timing.



Because AD and other similar offerings target mainly web applications a benchmark was created to measure the impact (drop) on the average throughput in the instrumentation and measurement of a servlet listed [below](#).

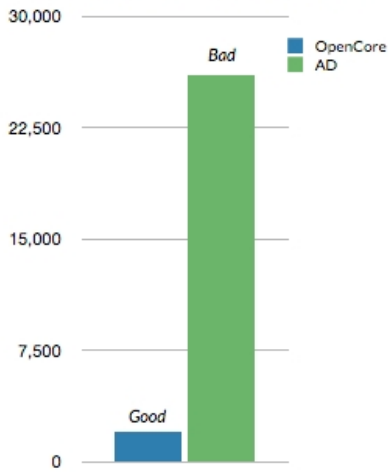
**Note:** A web archive containing the servlet was deployed to an Apache Tomcat 6.x container with Apache JMeter used to generate 1,000,000 HTTP GET requests.

Here is a chart comparing the throughput per minute using a single calling thread. AD creates a noticeably drag dropping down the throughput by more than 26,000 (13%) which again is very surprising considering that most of the request processing would be IO related, streaming requests and responses back and forth between client and container processes. JXInsight/OpenCore with its contextual [Servlet-to-URL](#) probes extension had less than 1% impact on the throughput.



The chart below provides a truer picture of the efficiency differences between OpenCore and AD in using the throughput delta (from the baseline).

Java Servlet Throughput Delta - Bigger is Worse



In summing up "low overhead" claims are not very useful without some context and clarification on what is the underlying behavior that is being measured, how it is being measured and to what degree. A much better approach is based on a cost efficiency comparative analysis of different approaches, configurations and solutions.

What the application performance management industry needs is a suite of benchmarks that accurately determine the unit cost for various measurement approaches offered by vendors which can then be run through models, equations even simulations to predict cost in terms of time and resource consumption on application runtimes and service. *This is becoming increasingly important with the cloud driving more and more pay-as-go like charges via metering.*

#### JDBC Benchmark

```
package jdbc;

import java.sql.*;
import java.util.concurrent.TimeUnit;

import static java.lang.System.nanoTime;
import static java.sql.DriverManager.getConnection;

public class Benchmark {

    static {
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws Throwable {
        long total = 0;

        for(int i=0; i < 10; i++) {
            Thread t = new Thread(new Runner());
            total -= nanoTime();
            t.start();
            t.join();
            total += nanoTime();
            System.out.println(TimeUnit.NANOSECONDS.toMillis(total));
        }
    }

    private static final class Runner implements Runnable{
        private static final String[] AIRPORTS =
            new String[]{"AMS", "HKG", "IST", "NRT", "SFO"};

        public void run() {
            for(int i=0; i < 10000; i++)
                iterate();
        }

        private static void iterate() {
            try {
                Connection connection = getConnection("jdbc:derby:benchmark");

                try {
                    count(connection, "select count(*) from cities");
                    count(connection, "select count(*) from flights");

                    list(connection, "select * from flights where orig_airport = ?", AIRPORTS);
                } finally {
                    connection.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

private static void count(Connection connection, String sql)
throws SQLException {
    final Statement stmt = connection.createStatement();
    try {
        final ResultSet rs = stmt.executeQuery(sql);
        try {
            if(rs.next())
                rs.getInt(1);
        } finally {
            rs.close();
        }
    } finally {
        stmt.close();
    }
}

private static void list(Connection connection, String sql, String[] ids)
throws SQLException {
    final PreparedStatement stmt = connection.prepareStatement(sql);
    final int count = ids.length;
    for(int i=0; i < count; i++) {
        stmt.setString(1, ids[i]);
        final ResultSet rs = stmt.executeQuery();
        try {
            while(rs.next()) {}
        } finally {
            rs.close();
        }
    }
}
}

```

*Note: The list () method has an "innocent" resource leak typical of todays applications but something that should not trouble an application performance management solution since the close () call on the connection will automatically trigger any necessary resource clean up.*

#### Servlet Benchmark

```

package com.acme;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;

public class Ping extends HttpServlet {
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    }
}

```

#### OpenCore Contextual Metering Screenshots

## Metering Tree

Context	Probe	Meter	Count
1515	<>	clock.time	*8,000,000
1515	java	clock.time	*8,000,000
1515	sql	clock.time	*8,000,000
1515	#url	clock.time	*8,000,000
1515	jdbc:derby:benchmark	clock.time	*8,000,000
1515	#sql	clock.time	7,600,000
1515	select count ( * ) from flights	clock.time	500,000
1515	#resultset	clock.time	300,000
1515	next	clock.time	100,000
1515	close	clock.time	200,000
1515	#statement	clock.time	200,000
1515	executeQuery	clock.time	100,000
1515	close	clock.time	100,000
1515	select * from flights where orig_airport = ?	clock.time	6,600,000
1515	#resultset	clock.time	6,100,000
1515	next	clock.time	5,200,000
1515	close	clock.time	900,000
1515	#statement	clock.time	500,000
1515	executeQuery	clock.time	500,000
1515	select count ( * ) from cities	clock.time	500,000
1515	#resultset	clock.time	300,000
1515	next	clock.time	100,000
1515	close	clock.time	200,000
1515	#statement	clock.time	200,000
1515	executeQuery	clock.time	100,000
1515	close	clock.time	100,000
1515	#connection	clock.time	400,000
1515	close	clock.time	100,000
1515	prepareStatement	clock.time	100,000
1515	createStatement	clock.time	200,000

Context	Probe	Meter	Count
1515	<>	clock.time	1,000,000
1515	javax	clock.time	1,000,000
1515	servlet	clock.time	1,000,000
1515	#protocol	clock.time	1,000,000
1515	http	clock.time	1,000,000
1515	#server	clock.time	1,000,000
1515	localhost	clock.time	1,000,000
1515	#port	clock.time	1,000,000
1515	8080	clock.time	1,000,000
1515	#context	clock.time	1,000,000
1515	/pinger	clock.time	1,000,000
1515	#path	clock.time	1,000,000
1515	/Ping	clock.time	1,000,000

### Software

JINSPiRED JXInsight/OpenCore 6.2.7.2

Apache Tomcat 6.0.32

JavaDB (Derby) 10.5.3.0

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