

William Louth's Weblog [Wordpress Archive]

The Art of API Design and Performance Engineering

Which Ruby VM? Consider Monitoring!

by williamlouth on March 17, 2011

There are a few negatives associated with benchmarking but in general benchmarks help engineering teams get time and resources from product management to focus on this important aspect of software. The continued innovation in VMs for languages such as Java, Ruby and JavaScript is a testament to this. But one thing that has always struck me as odd is that most of the non-microbenchmarks seem to ignore a fundamental concern of operational management – **production monitoring**. Benchmarks are designed and run completely oblivious to this. The stakes are high and it would seem the risks are even higher in benchmarks that are meant to be representative of real-world usage. In the context of Ruby VM selection this oversight can be detrimental to the performance and cost of monitored applications as there are significant differences in the underlying monitoring capabilities, the associated overhead, and tooling.

At the moment there are two major Ruby VM camps. Those VM's written in C/C++ versus a single one written in Java which itself is executed by a runtime written primarily in C/C++. Each camp has its own performance gains and losses but for the most part its a pretty close race except until it comes to production monitoring in which there is a clear winner.

Below is a Ruby class I am going to use to determine and compare the unit cost of New Relic, which is the most touted and overrated (*you'll soon see why*) monitoring solution for C/C++ based Ruby VM's though its written in Ruby, with JINSPIRED's OpenCore, which is the highest performing metering/monitoring solution in Java with unique native JRuby-to-Ruby integration.

```
class Loop
  def run()
    @starttime = Time.now
    for i in 1..10000000 do
      call
    end
    @endtime = Time.now
    puts (@endtime - @starttime)
  end

  def call()
  end
end

begin
  l = Loop.new
  for i in 1..10 do
    l.run
  end
end
```

New Relic offers two means in which to instrument a Ruby class for performance measurement and monitoring purposes. Here is our Ruby class sprinkled with instrumentation calls to the first of these. Both `Loop.call()` and `Loop.run()` are instrumented.

```
require 'rubygems'
require 'newrelic_rpm'

class Loop
  include NewRelic::Agent::MethodTracer

  def run()
    @starttime = Time.now
    for i in 1..10000000 do
      call
    end
    @endtime = Time.now
    puts (@endtime - @starttime)
  end

  add_method_tracer :run

  def call()
  end

  add_method_tracer :call
end

begin
  l = Loop.new
  for i in 1..10 do
    l.run
  end
end
```

The second way of collecting performance data with New Relic is using a transaction tracer. This generally collects much more detailed information related to invocation patterns than the simple metric aggregating method tracer though in our test case it has very little work to do because its only applied to the empty `Loop.call()` method.

This is the best, best case for such a collector. In fact when it comes to more contextual traces such as SQL queries the overhead is substantially higher but this overhead can be masked if your database is slow performing.

```
require 'rubygems'
require 'newrelic_rpm'

class Loop
  include NewRelic::Agent::Instrumentation::ControllerInstrumentation

  def run()
    @starttime = Time.now
    for i in 1..1000000 do
      call
    end
    @endtime = Time.now
    puts (@endtime - @starttime)
  end

  def call()
    end

  add_transaction_tracer :call
end

begin
  l = Loop.new
  for i in 1..10 do
    l.run
  end
end
```

Here are the New Relic “very best case” overhead costs per call invocation running the two instrumented versions of our Loop Ruby class on two different Ruby VM’s after subtracting the cost in executing a non-instrumented version of the Loop.call() method. Clearly your choice of instrumentation technique has a bearing on the winning Ruby VM though I would question whether any application could afford such excessive overhead.

It took less than 1 sec to perform a single Loop.run() call without any instrumentation. With transaction tracing it took 67 minutes!!!

microseconds	New Relic Method Trace	New Relic Transaction Trace
Ruby 1.8.7	21.431	220.814
JRuby 1.6.RC3	18.415	403.117

For the OpenCore comparison benchmark run no code changes are required. All that’s needed is the setting of the environment variable, JRUBY_OPTS, to

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

and VERIFY_JRUBY to true.

I performed two test runs with OpenCore’s metering engine. The first run had our dynamic cost aware hotspot strategy enabled, which is the default setting for production, and the second run with it disabled. The difference between both OpenCore test runs is that after a number of Loop.call() invocations the hotspot strategy would effectively disable all further metering of the instrumented method because of its runtime cost profile. The instrumentation would be still present but short-circuited.

microseconds	New Relic Method Trace	New Relic Transaction Trace	OpenCore Strategy Hotspot	OpenCore Strategy None
JRuby 1.6.RC3	18.415	403.117	0.005	0.204

The overhead difference is huge giving JRuby a significant lead over all Ruby VM implementations in being able to perform between 4,000 and 83,000 more OpenCore metered call

invocations for every **single** traced call invocation performed by New Relic.

OpenCore + JRuby offers Ruby developers deploying applications into production with greater code coverage, lower overhead, less risk, as well as savings in computing costs and resource consumption. More importantly it eliminates the guess work in deciding what should be instrumented and measured via the built-in cost awareness and self regulation of the metering runtime.

Here is a screen video showing how to use both the JRuby built-in profiler and OpenCore's JRuby-to-Ruby metering extension.

Software

JINSPIRED OpenCore 6.2.M7
New Relic RPM 2.13.5.beta4

```
java version "1.6.0_22"  
Java(TM) SE Runtime Environment (build 1.6.0_22-b04-307-10M3261)  
Java HotSpot(TM) 64-Bit Server VM (build 17.1-b03-307, mixed mode)
```

Environment

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: APPLICATION PERFORMANCE MANAGEMENT, OPENCORE,
RUBY
