

William Louth's Weblog [Wordpress Archive]

The Art of API Design and Performance Engineering

How to Meter Anything: Focus on Change

by williamlouth on April 28, 2011

Following my posting last week on exception cost metering I was asked how metering could be used to approximate a maximum stack depth for software service entry points. I could have easily used the `event` probes provider to listen in on the software execution recording the maximum delta between two counters on a per thread basis. One counter recording the number of `BEFORE_BEGIN` event call backs and another counter the number of `AFTER_END` events but I would still need a way to associate these with entry points which might not be known or fixed at the time of development. Whilst this approach is possible by way of the metering engine extension hooks it is not actually metering per se.

But how can the depth of a call stack be modeled as a meter when meters are basically thread specific cumulative counters and the depth of a call stack goes up and down. Once we hit the maximum stack depth the meter would never change and thus no further metering assignment would be made against those activities on the metering stack that did have a stack depth but not one that exceeded a previous maximum recorded for the thread. The trick is not to see the current value of a meter as the stack depth – but the delta from some starting point.

Metering is primarily focused on the change that occurs between the beginning and ending of a metered activity. The meter readings themselves are largely irrelevant – at most secondary. What is required is that the maximum stack depth be tracked within the current execution context of a service entry point call.

Here is how this can be achieved using OpenCore's optional `interceptor` probes provider and its Open API extension point interfaces. The code only ever increases the underlying value of the resource measure registered, accessed by the `getValue()` method, when the stack depth has been exceeded. The current `depth` in terms of metered activities exceeds the `max` recorded. Both the `depth` and `max` are reset following the completion of the outer most method which is the only method that is metered as indicated by the boolean expression in the `return` statement of the `begin()` method. The `value` field in fact represent the number of changes (increments) in the `max` field within and across multiple outer most method invocations.

```
package com.jinspired.opencore.probes.meter.stack.depth;

import com.jinspired.jxinsight.probes.Probes;
import com.jinspired.jxinsight.probes.interceptor.ProbesInterceptor;
import com.jinspired.jxinsight.probes.interceptor.ProbesInterceptorClosure;
import com.jinspired.jxinsight.probes.interceptor.ProbesInterceptorContext;
import com.jinspired.jxinsight.probes.interceptor.ProbesInterceptorFactory;

import static com.jinspired.jxinsight.probes.Probes.name;

public final class InterceptorFactory
    implements ProbesInterceptorFactory {

    private static final Probes.Name NAME = name("stack").name("depth");

    public void init() { }

    public ProbesInterceptor create(Probes.Name n,
        ProbesInterceptorClosure cl) {
        Interceptor i = (Interceptor) cl.get();

        if(i == null) {
            cl.set(i = new Interceptor());
            Probes.get().register(NAME, i);
        }

        return i;
    }

    private static final class Interceptor
        implements ProbesInterceptor, Probes.Measure {

        private long value;
        private int depth; // within the current execution context
        private int max; // within the current execution context

        public boolean begin(ProbesInterceptorContext ctx,
            ProbesInterceptorClosure cl) {

            int d = depth;

            // reset the max if at an entry point
            if(d == 0)
                max = 0;

            depth = ++d;

            if(d > max) {
                value++;
                max = d;
            }

            return d == 1; // only meter entry points
        }

        public void end(ProbesInterceptorContext ctx,
            ProbesInterceptorClosure cl,
            boolean begun) {
            --depth;
        }

        public long getValue() {
            return value;
        }
    }
}
```

Here are the system properties I added to a `jxinsight.override.config` file to install the extension point and map the registered resource measure to a meter named "stack.depth".

```
jxinsight.server.probes.strategy.enabled=false
jxinsight.server.probes.interceptor.enabled=true
jxinsight.server.probes.interceptors=stack.depth
jxinsight.server.probes.interceptor.stack.depth.factory.class=
com.jnspired.opencore.probes.meter.stack.depth.InterceptorFactory
jxinsight.server.probes.meter.resources=stack.depth
```

Below is a metering model collected following the completion of a JBoss 6.0 server startup and some simple web request to their noticeably slow responding web based mgmt console. Only the outer most method invocations are metered with the stack depth recorded for those methods not metered but called directly and indirectly from such metered points. From the metering model we can see that greatest stack depth recorded was 120 and this was during the invocation of `process()` method listed first. The minimum of 48 is smallest maximum stack depth for an invocation of the `process()` method.

Note: A benefit in modeling the stack depth as a meter is that we get additional statistical data including min, max and avg out of the box and that is before we enable more exotic statistical measurements afforded by OpenCore's metering runtime which by default is configured with a bare-metal and extremely efficient runtime stack.

Metering Table					
Probe	Meter	Min	Avg	Max	
o.a.c.h.Http11Protocol\$Http11ConnectionHandler.process	stack.depth	*48	*88	*120	
o.r.c.p.i.RuntimeDiscoveryExecutor.call	stack.depth	29	32	35	
o.r.c.p.i.AutoDiscoveryExecutor.call	stack.depth	32	32	32	
o.r.c.p.c.ContentDiscoveryRunner.call	stack.depth	30	30	30	
o.j.j.HypersonicDatabase.startService	stack.depth	29	29	29	
o.j.e.p.k.h.HiLoKeyGeneratorFactory.startService	stack.depth	27	27	27	
o.r.c.p.u.DiscoveryComponentProxyFactory\$ComponentI...	stack.depth	1	6	25	
o.r.c.p.i.ResourceContainer\$ComponentInvocationThre...	stack.depth	1	3	25	
o.h.j.jdbcDatabaseMetaData.getTables	stack.depth	24	24	24	
c.a.a.a.t.o.m.ObjStoreBean.getObjectStoreBrowserBean	stack.depth	22	22	22	
o.h.c.r.i.n.NettyAcceptor.startServerChannels	stack.depth	21	21	21	
o.j.w.t.s.d.TomcatDeployment.performDeploy	stack.depth	19	20	21	
o.h.c.s.m.i.ManagementServiceImpl.sendNotification	stack.depth	11	13	18	
o.j.r.c.InternalManagedConnectionPool.fillToMin	stack.depth	18	18	18	
c.a.a.j.j.RecoveryManagerService.create	stack.depth	17	17	17	

Typically one wants to limit the metering until a particular method appears on the stack representing the actual service rather than the underlying request dispatching middleware. This can be achieved using the `entrypoint` probes provider along with the custom extension point. The following additional properties ensure that metering does not occur until a method is called within the `org.jboss.seam` package or its sub-packages.

```
jxinsight.server.probes.entrypoint.enabled=true
jxinsight.server.probes.entrypoint.name.groups=org.jboss.seam
```

And here is the revised metering model. I did say that **anything** can be metered!! All that is required is a change in thinking.

Metering Table					
Probe	Meter	Min	Avg	Max	
o.j.s.s.SeamFilter.doFilter	stack.depth	20	*50	*104	
o.j.s.s.SeamListener.contextInitialized	stack.depth	*28	28	28	
o.j.s.i.BeanInterceptor.invoke	stack.depth	26	26	26	
o.j.s.s.SeamFilter.init	stack.depth	18	18	18	
o.j.s.j.SeamApplicationFactory.getApplication	stack.depth	12	12	12	
o.j.s.l.Logging.getLogProvider	stack.depth	7	7	7	
o.j.s.j.SeamApplication.getELResolver	stack.depth	5	5	5	
o.j.s.j.SeamApplication.addConverter	stack.depth	2	2	3	
o.j.s.i.m.d.SeamUrlIntegrationDeployer.<init>	stack.depth	2	2	2	
o.j.s.j.SeamApplication.setActionListener	stack.depth	2	2	2	
o.j.s.j.SeamApplication.setNavigationHandler	stack.depth	2	2	2	

FROM: CODE EXECUTION ANALYSIS, METERING, OPENCORE, PERFORMANCE ENGINEERING, PERFORMANCE TESTING

