Dynamic Tracing Framework for Oracle WebLogic Server on Oracle Solaris
Executive Overview

This paper outlines a new framework combining Java BTrace and Solaris DTrace technologies to enable an extremely light-weight, powerful tool that provides deep observability into Oracle WebLogic server applications running on Oracle Solaris. The paper includes examples of the kind of data that can be extracted from a production system along with the D-scripts that make this possible.

Introduction

Tools for observing and correcting performance issues have traditionally focused on the development phase rather than the deployment phase. This means performance issues seen in production have to be recreated by developers outside of production before they could be addressed. A few tools have attempted to provide observability in production but they are limited to observing only a select few and often inadequate set of metrics.

This paper outlines a framework that provides a virtually unlimited amount of observability into Java applications in production. With easy to use interfaces and very low overhead, the framework is well suited for both developers and system administrators to quickly answer complicated performance questions and to take corrective action directly on production systems. Along with the other tools available on Oracle Solaris, this framework makes Oracle Solaris the best platform on which to run Oracle WebLogic Server in development and production.

Along with the other tools available on Oracle Solaris, this framework makes Oracle Solaris the best platform on which to run Oracle WebLogic Server in development and production.
Dynamic Tracing Framework for Oracle WebLogic Server on Oracle Solaris

Observing Applications in Production

With the advent of multi-tier architectures, today’s applications have become very complex. Each of the application tiers has excellent tools for observability and debugging, but observing applications across these multiple tiers presents many challenges. This problem becomes even more complicated if you want to observe these applications in production. Applications in production are sensitive to performance impacts. It is not always easy to stop and start applications to enable debug flags. Adding debug versions of applications into production involves expensive and time-consuming quality assurance (QA) cycles. All these issues complicate the problem of observation. Without good observation, performance tuning becomes a trial and error guessing game.

The few traditional tools that try to address these multi-tier observation problems have many limitations and often are inadequate in production. Such tools often limit observation to pre-defined metrics. As these metric are defined at the time of observation tool development, the tool developer decides the metrics that would be collected. In many cases these metrics are not sufficient to fully understand application behavior. Many of these tools are also expensive and require complicated install and setup. Because of these limitations, sometimes the only way to collect the metrics is by adding custom code into applications, making the process laborious and time consuming.

Solaris DTrace.

DTrace, a Dynamic Tracing framework was developed to address this very problem. DTrace can be used to observe all tiers of the application stack. It is truly dynamic and does not require application code changes or even an application restart. Fully optimized applications can be observed using DTrace. The overhead of observation is low and there is no overhead when observation is turned off. Instrumentation can be turned on and off dynamically, so you only incur the cost of collecting information when it is needed. DTrace is safe and turns itself off when observation overhead affects system performance.

DTrace can be used to observe applications developed in C, C++, Java, Javascript, Ruby, PHP, Perl, and python among other programming and scripting languages. Several database products have built in support for DTrace. Other system layers, such as I/O, networking, application and kernel locks, and CPU counters, can also be observed using DTrace. DTrace has been one of the hallmark feature of Solaris 10 since its release in 2005.

DTrace basics

D-scripts are used to enable and program points of instrumentation. The D-script format does not change based on the application tier being observed; and a single script can be used to observe multiple tiers at the same time. Details of the D scripting languages are beyond the scope of this white paper. Please see the resource section for some DTrace links.

Next, we explain some basic DTrace concepts.
DTrace can instrument any location in a live running application. In DTrace, points of instrumentation are called \textbf{probes}. D-scripts use the D language to enable probe points. The D-scripts format is explained in Figure 1.

![Figure 1. D-scripts](image)

- **Probes** are defined using 4-tuples, \texttt{provider:module:probe:name}. This divides the name space to easily identify instrumentation location(s).
- **Actions** are callbacks that are called when the probe points are executed (or fired). If the action list is empty, dtrace inserts a \texttt{printf()} for your convenience. The action section can collect any arbitrary information from the probe point. Actions are fully programmable.
- **Predicates** allow one to limit when actions are executed. A predicate is a D expression that must evaluate to true for the actions to execute i.e., predicates are used like “if” statements.

### Observing Java Applications using DTrace

DTrace can be used to observe Java applications running in JVM versions 1.4.2 and above. The following parts of the JVM can be instrumented using DTrace.

- Garbage collection
- Thread creation
- Object Allocation
- JVM lifecycle
- Classloading and unloading
- Method compilation
- JNI

Method invocations can also be observed using DTrace even though this tends to cause a higher instrumentation overhead since DTrace has only one instrumentation point for all method invocations. As a result, all method invocations need to be instrumented to be able to observe a single method. In
Java EE applications, the number of methods invoked can be extremely high, so the overhead of observation can be pretty high.

BTrace - Java byte code instrumentation

BTrace is a framework for the dynamic tracing of Java applications. BTrace can selectively instrument any method in a live running Java application. Similar to DTrace, the enabling and disabling of instrumentation is truly dynamic. This means that neither application code changes nor application restart is required and even fully optimized applications can be instrumented. BTrace has its own ‘Java like’ language for enabling and programming instrumentation points. BTrace uses byte code instrumentation to enable dynamic tracing. In addition, BTrace uses published interfaces for instrumentation, so it can be used with all standard Java implementations.

Framework combining BTrace and DTrace

Combining BTrace and DTrace provides numerous advantages. BTrace can selectively enable instrumentation in Java, thus greatly reducing the overhead of dynamically tracing Java applications. Marrying DTrace and BTrace provides the ability to correlate Java code to other parts of the application and to the operating system stack. Having a single language for creating and programming instrumentation simplifies the usage of both tools. A wide variety of tools that use DTrace can now be used to observe Java applications. Furthermore, since application deployments today are complex often running multiple copies of the JVM on the same system, a combination of DTrace and BTrace can be used observe the interaction between multiple JVMs.

Combining the two technologies is an excellent concept, however there are several issues that must be addressed before this approach can be used in practice. First, the process of instrumentation must be simplified. Second, the entire process needs to be automated as much as possible. This automation must not change the D-script format. This will ensure that tools built for DTrace will work with the new framework.
Dynamic Tracing Framework for Oracle WebLogic Server

The Dynamic Tracing Framework for WebLogic Server (DFrame) has been developed with these goals in mind. DFrame is simple to install, it automates the instrumentation process and it preserve the D-script format allowing the current DTrace tool chain to be leveraged. DFrame uses a simple flat file interface and so is extremely nimble to use and easy to deploy.

Internally, DFrame uses BTrace, DTrace and other published and well understood technologies. So, DFrame will work on any JVM that supports BTrace on Oracle Solaris 10. DFrame is precise enough to produce really fine grained targeted instrumentation and yet it is flexible enough to easily instrument the entire application.

This white paper documents the process of installing and deploying DFrame, as well as presents a few real life case studies. Please note that this tool is not yet released. Please email dframe-interest_us@oracle.com if you would like pre-release access to DFrame.

DFrame Architecture

Figure 2 shows the DFrame architecture. DFrame is controlled by the config file embedProbes.conf, which contains the instrumentation details. The embedProbes command reads the configuration file and instruments the WebLogic Server. The telemetry from this instrumentation can be consumed using D-scripts.
Using DFrame

Prerequisites

DFrame has been tested on both the 32 and 64-bit JDK versions running on Oracle Solaris 10 SPARC and x86 platforms.

By default, DTrace requires root access on the system. A system administrator can provide DTrace privileges to regular non-root accounts using the Solaris Least privilege model. Running the D-scripts in DFrame requires at least dtrace_proc privilege. See http://www.solarisinternals.com/wiki/index.php/DTrace_Topics_Guide#Privileges for details on adding privileges.

DFrame works well in Solaris Zones. The Zones needs to have DTrace enabled using the steps described in http://docs.sun.com/app/docs/doc/819-2450/6n405mdtc?l=en&ca=view

Installing DFrame

DFrame can be installed in any directory. Copy the DFrame package into a directory.

Change directory into the directory

```bash
# cd <install dir>
```

Untar

```bash
# tar xf DFrame.tar
```

Set the path and other environment variables for Dframe

```bash
# cd dframe
# source setpath (or . ./setpath)
```
Steps to Use DFrame to collect performance data

Using DFrame is simple. There is a single command, embedProbes. In its simplest form embedProbes just takes the process id of the WebLogic (or Java) process to be instrumented. This command reads the configuration file embedProbes.conf and instruments the WebLogic server instance. Once instrumented D-scripts can be developed to collect and process telemetry from the instrumentation. The embedProbes command needs to be run as the user that runs WebLogic. Make sure that this user has write permission on the DFrame directory.

The embedProbes command is run as follows

```
% embedProbes 1234
```

This command reads the embedProbes.conf file, creates a BTrace script based on the file, compiles the resulting BTrace script and then embeds it into the Java process listed (here process id 1234 is the process that DFrame is targeting). All this is accomplished in one simple step. To remove the instrumentation just kill the embedProbes command by pressing Control-c.

The embedProbes command also understands the followings options.

- `-f` use a custom configuration file instead of embedProbes.conf
- `-B` save the generated BTrace script as EmbedProbes.java
- `-d` print debug messages
- `-h` print the usage message

The embedProbes.conf file

DFrame is controlled by the embedProbes.conf configuration file. This is a simple flat file, located in the root directory of DFrame. This file can be edited to include the location where probes need to be embedded. The framework supports two types of probes, mapped and unmapped probes.

**Mapped Probes**

A mapped probe assigns a logical name to a probe. For example, the invocation of the method weblogic.work.ExecuteThread() can be mapped to a logical probe, called work-start and the return from that method may be mapped to a logical probe called work-end. Multiple method invocations can be mapped to the same logical probe and multiple logical probes can be mapped to the same method invocation.

A single line in embedProbes.conf with the following format will create a logical probe:

```
classname:methodname:location:logicalname
```
The probe signature for mapped probes is `weblogic<pid>:::event`. In the D-script the logical probe name can be retrieved from arg0 and arg1 will have the name of the invoked class/method in the format, `classname.method`. For example the following D-script prints out the names of all the logical probes that fire.

```
#!/usr/sbin/dtrace -qs
weblogic*:::event
{
    printf("%s\n",copyinstr(arg0));
}
```

Unmapped Probe

An unmapped probe is a probe that fires when methods are invoked. An unmapped probe is created by listing the class name and the methods of interest in the `embedProbes.conf` file. These probes fire when the selected methods are invoked. For each method of interest, a new entry can be added to `embedProbes.conf` using the following format.

```
classname : methodname
```

This entry will create two probes of the format `weblogic<pid>:::method-entry` and `weblogic<pid>:::method-return`. As the name describes it, these probes will fire when the listed method is invoked and when it returns from its invocation, respectively. In the D-script the name of the class and method can be retrieved from arg0 and arg1 respectively.

Here is an example D-script that prints the name of the method and class for every method that is invoked. This is script to demonstrate the concept and it is not recommended for production use as the overhead may be substantial.

```
#!/usr/sbin/dtrace -qs
weblogic*:::method-entry
{
    printf("%s.%s\n",copyinstr(arg0),copyinstr(arg1));
}
```

Other `embedProbes.conf` formatting details

Both the classname and the method name fields understand wild cards ‘*’. This allows for specifying a wide set of methods in a single line.

DFrame also understands interfaces. A ‘+’ preceding the class name will be interpreted as an interface name. This will instrument every class that implement that interface.
The embedProbes.conf file uses # to denote comments. Any characters following the # will be ignored until the end of line is reached.

Sample embedProbes.conf file

Here is a sample embedProbes.conf file:

```plaintext
javax/servlet/http/HttpServlet:service:entry:http-start #mapped probe example
+javax/servlet/http/HttpServletRequestWrapper:getMethod:entry:httpget-start
    # interface
*medrec*Patient*::* # unmapped probes & wild card example
```

DFrame ships with a few sample config files. The samples include configuration files for standard Java EE interfaces as well as some specific to the MedRec sample application described in the next section.

Use case - The MedRec Workload

The DTrace Framework for WebLogic Server is demonstrated using the MedRec, a sample application that ships with Oracle WebLogic Server.

The MedRec Application

Avitek Medical Records (MedRec), is an Oracle WebLogic Server sample application suite that demonstrates many aspects of the Java EE platform. MedRec was designed as an educational tool for Java EE developers. It showcases the use of Java EE components, and illustrates best practice design patterns for component interaction and client development.

The application itself provides a framework for patients, doctors, and administrators to manage patient records using a variety of different clients. There are two types of patient records that are managed by MedRec:

**Patient profile information**—Consists of a patient’s name, address, social security number, and login information.

**Patient medical records**—Consists of details about a patient’s visit with a physician such as the patient’s vital signs and symptoms as well as the physician’s diagnosis and prescriptions.

The application suite consists of two main Java EE applications. Each application supports one or more user scenarios for MedRec:

**medrecEar**—Patients log into the patient Web Application (patientWebApp) to edit their profile information, or request that their profile be added to the system. Patients can also view prior medical records of visits with their physician. Administrators use the administration Web Application (adminWebApp) to approve or deny new patient profile requests. medrecEar also provides all of the
controller and business logic used by the MedRec application suite, as well as the Web Service used by different clients.

**physicianEar**—Physicians and nurses log into the physician Web Application (physicianWebApp) to search and access patient profiles, to create and review patient medical records, and to prescribe medicine to patients. The physician application is designed to communicate using the Web Service provided in medrecEar.

The medrecEar and PhysicianEar applications are deployed to an Oracle WebLogic Server instance called MedRecServer. The physicianEar application communicates with the controller components of medrecEar by using Web services.

**The MedRec Workload**

A workload was created to drive the MedRec application in order to have the capability to vary the load to test different scenarios.

The workload executes the following sequence of operations:

1. New Patient registers.
2. The registration is approved by Admin user
3. Physician searches for a patient
4. A Patient updates his profile.
DFrame Case studies

This section will cover a few case studies of using DFrame in real life. All the configuration files listed in this section are bundled with the DFrame package and the source code and explanation of the D-scripts are included in a “Detailed Description of Bundled Scripts” section of this white paper.

What are the most expensive methods in my application?

One key step in profile the runtime execution of an application is determining the list of most frequently executed methods and which methods take the most time to execute. DFrame can be used to identify these methods very easily. In this use case, the entire MedRec application is instrumented. The flexibility of the `embedProbes.conf` file permits this level of exhaustive instrumentation with a single line.

`embedProbes.conf`

```
*medrec*: *
```

Two D-scripts, `count_methods.d` and `method_times.d` are used for this purpose. The `count_methods.d` script is pretty simple.

```
#!/usr/sbin/dtrace -qs
weblogic*:::method-entry
{
    @([copyinstr(arg0),copyinstr(arg1)]=count();
}
END
{
    printf("%-60s %-20s %10s
","CLASS","METHOD","COUNT);
    printa("%-60s %-20s %@10s
",@);
}
```

Results

Here is the output from the scripts. We only show a few lines for brevity.

```
<table>
<thead>
<tr>
<th>CLASS</th>
<th>METHOD</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.bea.medrec.actions.AdminLoginAction</td>
<td>authenticate</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.actions.AdminLogoutAction</td>
<td>executeAction</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.actions.ApprovePatientRequestAction</td>
<td>approve</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.actions.ApprovePatientRequestAction</td>
<td>composeMail</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.entities.UserEJB</td>
<td>&lt;init&gt;</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.entities.UserEJB</td>
<td>ejbCreate</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.entities.UserEJB</td>
<td>ejbPostCreate</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.value.Mail</td>
<td>&lt;init&gt;</td>
<td>227</td>
</tr>
</tbody>
</table>
```
Understanding the output

Here is how to look at the output. The first column is the class name, the second column is the method name. The last column has the number of times the method was invoked. Just looking at this output can provide some insight into the application. For example from the output we see that the many of these methods were invoked 227 times. This reflects that the transactions were executed 227 times. It is equally interesting to note that the getAddress() method was called 9761 times. That is almost 36 times per transaction. So this is one area to consider for optimization.

Note that DTrace, by default, prints data in the ascending order. This ensures that the most interesting data is printed last and so there is no need to scroll back to see the information on the terminal. The "-x aggsortrev" option can be used in any of the scripts to reverse the sort order.

The TimeSpent.d script is also relatively simple. Here is the script.

```bash
#!/usr/sbin/dtrace -qs
weblogic*:method-entry
{ this->cn = copyinstr(arg0);
  this->mn = copyinstr(arg1);
  self->ts[this->cn, this->mn]=timestamp;
}
weblogic*:method-return
{ this->cn = copyinstr(arg0);
  this->mn = copyinstr(arg1);
}
weblogic*:method-return
/self->ts[this->cn, this->mn]/
{ @s[this->cn, this->mn]=sum(timestamp - self->ts[this->cn, this->mn]);
  @c[this->cn, this->mn]=count();
  self->ts[this->cn, this->mn]=0;
}
```
Here is the output from the `TimeSpent.d` script.

<table>
<thead>
<tr>
<th>CLASS_METHOD</th>
<th>TIME(ns)</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.bea.medrec.value.Mail.setMessage</td>
<td>1393129</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.value.Mail.setSubject</td>
<td>1399664</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.value.Search.&lt;init&gt;</td>
<td>1404939</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.value.Mail.&lt;init&gt;</td>
<td>1447305</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.value.User.&lt;init&gt;</td>
<td>1453135</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.beans.SearchBean.setSsn</td>
<td>1529311</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.value.Patient.setSsn</td>
<td>8322986</td>
<td>1284</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getState</td>
<td>8340586</td>
<td>1284</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.setSsn</td>
<td>8357920</td>
<td>1284</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.setCity</td>
<td>8384502</td>
<td>1284</td>
</tr>
<tr>
<td>com.bea.medrec.beans.AddressBean.getState</td>
<td>10594263</td>
<td>1498</td>
</tr>
<tr>
<td>com.bea.medrec.beans.AddressBean.getCountry</td>
<td>10642060</td>
<td>1498</td>
</tr>
<tr>
<td>com.bea.medrec.beans.AddressBean.getZipCode</td>
<td>10644243</td>
<td>1498</td>
</tr>
<tr>
<td>com.bea.medrec.beans.BaseBean.toStr</td>
<td>10718818</td>
<td>1284</td>
</tr>
<tr>
<td>com.bea.medrec.util.ServiceLocator.getObj</td>
<td>56418500</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.entities.PatientEJB_lbcar4_ELOImpl.setSsn</td>
<td>56993073</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.entities.PatientEJB_lbcar4_ELOImpl.setPhone</td>
<td>57231590</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.entities.AddressEJB_p2zm36_ELOImpl.setCity</td>
<td>57259660</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.actions.SearchResultsAction.processSearch</td>
<td>3933106887</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.actions.SearchResultsAction.search</td>
<td>3965563909</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.actions.PhysBaseLookupDispatchAction.execute</td>
<td>3977738731</td>
<td>214</td>
</tr>
<tr>
<td>com.bea.medrec.actions.BaseAction.execute</td>
<td>5447231518</td>
<td>3638</td>
</tr>
<tr>
<td>com.bea.medrec.filters.RequestEncodingFilter.doFilter</td>
<td>13326283764</td>
<td>4280</td>
</tr>
</tbody>
</table>

This is the output for a 2 minute run. It is clear from the results that `doFilter()` takes 13.3 sec or 11% of the run time was spent on this method. Again, a good candidate for optimization. Please note that the time spent is listed in nanoseconds and is inclusive of wall clock times.

As these results demonstrate, DFrame can be used to easily pinpoint prime candidates for optimization. In addition, the overhead to run these scripts are low, typically around 5% and so can be used in production. It is also good to remember that this overhead is only incurred when you are collecting and processing telemetry.
What part of the application is slowing down HTTP Requests?

Dynamic runtime execution profiling can also be used to answer the questions, “What part of the MedRec code is called to service the HTTP request?” and “How much time is spent in each of the methods for the request?” The DFrame framework makes it easy to obtain this information.

To achieve this, `embedProbes.conf` will contain the following three lines.

```
*medrec*::*
javax/servlet/http/HttpServlet:service:entry:http-start
javax/servlet/http/HttpServlet:service:return:http-end
```

The first line is a unmapped coarse grain probe that instruments all the methods of all the classes that have the word `medrec` in it. This one line will instrument all of the methods in the MedRec application. The last two lines are mapped probes. These are the logical HTTP entry and exit points.

The application is instrumented by issuing the `embedProbes` command.

```
./embedProbes 1234
```

where 1234 is the process id of the Oracle WebLogic Server process.

The `http-methods.d` D-script can be used to find out the methods invoked by HTTP request. This script lists the number of times and amount of time spent in MedRec methods to service the HTTP request.

### Results

```
./http-methods.d
CLASS, METHOD TIME(ns) COUNT
com.bea.medrec.value.Patient.setPhone 7764217 1228
com.bea.medrec.value.Address.setCity 7772288 1228
com.bea.medrec.value.Patient.getSsn 7814521 1228
com.bea.medrec.value.Patient.setEmail 7817248 1228
com.bea.medrec.value.Address.setCity 7822853 1228
com.bea.medrec.value.Address.setState 7845840 1228
com.bea.medrec.value.Address.setStreetName2 8343559 1228
com.bea.medrec.value.Address.setStreetName1 8397772 1228
com.bea.medrec.value.Address.getStreetName1 8423503 1228
com.bea.medrec.value.Patient.getMiddleName 8430561 1228
com.bea.medrec.beans.PatientBean.getDob 16689994 2460
com.bea.medrec.beans.PatientBean.toUser 17364993 205
com.bea.medrec.beans.PatientBean.getMiddleName 17699911 2460
com.bea.medrec.beans.BaseBean.getId 17783028 2665
com.bea.medrec.beans.PatientBean.getFirstName 17809213 2460
com.bea.medrec.beans.PatientBean.getLastName 17980862 2460
com.bea.medrec.actions.SearchResultsAction.processSearch 4212356200 205
com.bea.medrec.actions.SearchResultsAction.search 4261589358 205
com.bea.medrec.actions.PhysBaseLookupDispatchAction.execute 4277899374 205
com.bea.medrec.actions.BaseAction.execute 7019780498 3485
com.bea.medrec.actions.BaseLookupDispatchAction.execute 9904999643 1230
com.bea.medrec.filters.RequestEncodingFilter.doFilter 15987611900 4096
```
This was for a two minute run. It is interesting to note that the top six methods here take about 45.7 seconds or over 38% of the time and so are clearly methods to consider for optimization.

DFrame provides an easy way to quickly get to the most expensive methods. The http-method.s script can be modified to look for any Java EE event or even any method of interest.
What is the call flow for the most expensive method?

After expensive methods are identified, the next analysis step is to determine why they take so much time. To that end, one must track down the methods that are called by this expensive method, as well as its call flow pattern. DFrame can also be used to get this type of detailed information. Methods often have different call flow patterns. It is interesting to find the call flow patterns and the amount of time spent in each of the pattern. This type of analysis typically provides insight into why certain hot method take so much time to get to the root cause of performance issues.

For this example the embedProbes.conf file has a simple single line entry

```
*medrec*:
```

The call flow analysis D-script, cf-gen.d, takes a name of a class.method as its only input and prints the call flow for that method and the time spent in each of the called methods.

Results

```
# ./cf-gen.d "AdminLoginAction.authenticate"
->com.bea.medrec.actions.AdminLoginAction.authenticate
->com.bea.medrec.beans.UserBean.getUsername
<-com.bea.medrec.beans.UserBean.getUsername
->com.bea.medrec.beans.BaseBean.<init>
<-com.bea.medrec.beans.BaseBean.<init>
->com.bea.medrec.beans.AdminBean.<init>
<-com.bea.medrec.beans.AdminBean.<init>
->com.bea.medrec.actions.BaseAction.getRedirectPage
->com.bea.medrec.utils.MedRecWebAppUtils.isNotEmpty
<-com.bea.medrec.utils.MedRecWebAppUtils.isNotEmpty
<-com.bea.medrec.actions.BaseAction.getRedirectPage
<-com.bea.medrec.actions.AdminLoginAction.authenticate
Total Time: 4258835975 :: Fastest: 11305529 :: Slowest: 30815623 :: Average: 20673961 :: Count: 206

SUMMARY of all methods called from AdminLoginAction.authenticate
COUNT TOTAL CLASS.METHOD
206 41421278 com.bea.medrec.beans.AdminBean.<init>
206 808571271 com.bea.medrec.beans.BaseBean.<init>
206 934000182 com.bea.medrec.utils.MedRecWebAppUtils.isNotEmpty
206 1084972645 com.bea.medrec.beans.UserBean.getUsername
206 1313347802 com.bea.medrec.actions.BaseAction.getRedirectPage
206 4258835975 com.bea.medrec.actions.AdminLoginAction.authenticate
```

Observations

A closer look at the above output provides some interesting insights into the AdminLoginAction.authenticate() method. It takes about 4.2 seconds of the entire run. It was invoked 209 times for an average per invocation time of just over 20 milliseconds. But a closer look reveals that some invocations took just 10 milliseconds, while some took over 30 milli seconds. Also it is interesting to note that the method getRedirectPage() takes about 30% of the time to authenticate the login.

This seems pretty high and needs further study. One way to proceed would be run the cf-gen.d script on the getRedirectPage() method to get to the root cause of the problem. This is out of scope for this paper and is left as an exercise to the reader.
What is the call flow pattern for Java EE events?

A modified version of the `cf-gen.d` script can be used to collect information about call flow of the Java EE events. This script, `eventflow.d`, is also part of DFrame package.

For the next example, the `embedProbes.conf` will just contain logical Java EE probes.

```
# ./eventflow.d http
->javax.servlet.http.HttpServlet.service
->com.bea.medrec.actions.BaseAction.execute
<--com.bea.medrec.actions.LogoutAction.executeAction
<--com.bea.medrec.actions.BaseAction.execute
<-javax.servlet.http.HttpServlet.service
Total Time: 20180373 :: Count: 6

->javax.servlet.http.HttpServlet.service
->com.bea.medrec.actions.BaseAction.execute
<--com.bea.medrec.actions.AdminLogoutAction.executeAction
<--com.bea.medrec.actions.BaseAction.execute
<-javax.servlet.http.HttpServlet.service
Total Time: 26204638 :: Count: 7

->javax.servlet.http.HttpServlet.service
->com.bea.medrec.actions.BaseLookupDispatchAction.execute
->com.bea.medrec.actions.EditProfileAction.defaultMethod
<--com.bea.medrec.actions.EditProfileAction.defaultMethod
<--com.bea.medrec.actions.BaseLookupDispatchAction.execute
<-javax.servlet.http.HttpServlet.service
Total Time: 41377187 :: Count: 6

->javax.servlet.http.HttpServlet.service
->com.bea.medrec.actions.BaseAction.execute
->com.bea.medrec.actions.SearchAction.executeAction
<--com.bea.medrec.actions.SearchAction.executeAction
<--com.bea.medrec.actions.BaseAction.execute
<-javax.servlet.http.HttpServlet.service
Total Time: 58970456 :: Count: 6

->javax.servlet.http.HttpServlet.service
->com.bea.medrec.actions.BaseAction.execute
->com.bea.medrec.actions.ViewProfileAction.executeAction
<--com.bea.medrec.actions.ViewProfileAction.executeAction
<--com.bea.medrec.actions.BaseAction.execute
<-javax.servlet.http.HttpServlet.service
Total Time: 77373233 :: Count: 12
```
Observations

As the output clearly shows the HTTP request can take multiple code paths depending on the request. The output is sorted by the time spent in each code path. As can be clearly seen from the output, HTTP requests times vary from 20 milliseconds to 77 milliseconds depending on the actions that are executed. A deeper study would be to use the `cf-gen.d` script to look into interesting actions. The `embedProbes.conf` file can be varied to get more details into the call flow. This further drill down is left as an exercise for the reader.

This section provides some ideas and tools for using DFrame to answer complex questions.
Why does Admin Approval take so long?

During load test, an interesting problem was observed. The logs provided by the load generator indicate that the time taken by Admin to approve a new Patient registration is very high compared to other operations. This can be seen in the following graph:

The Admin Patient Approval process takes 10 times as long as other processes.

Step one in understanding this issue is to profile “Admin Patient Approval”. The `TimeSpent.d` script that was discussed earlier can be used for this purpose. DFrame provides the ability to selectively instrument Java methods. This permits instrumenting and monitoring all the ‘actions’ in MedRec. The config file will be a simple one liner.

```
embedProbes.conf
*action*::*
```

The output from the `TimeSpent.d` script with the above config file.

```
# ./TimeSpent.d
com.bea.medrec.actions.BaseAction.getRedirectPage 12028549 675
com.bea.medrec.actions.ApprovePatientRequestAction.composeMail 14674664 225
com.bea.medrec.actions.BaseLookupDispatchAction.setupLocale 14818675 225
com.bea.medrec.actions.LoginAction.authenticate 621716175 225
com.bea.medrec.actions.LoginAction.executeAction 691750580 450
com.bea.medrec.actions.RegisterAction.register 712973251 225
com.bea.medrec.actions.ApprovePatientRequestAction.approve 805175108 225
com.bea.medrec.actions.ViewRequestsAction.executeAction 1021986656 450
com.bea.medrec.actions.EditProfileAction.save 1137907672 225
com.bea.medrec.actions.SearchResultsAction.processSearch 3321604112 3825
com.bea.medrec.actions.SearchResultsAction.search 4669957029 225
com.bea.medrec.actions.PhysBaseLookupDispatchAction.execute 4705870772 225
org.apache.struts.actions.DispatchAction.dispatchMethod 7397950773 900
org.apache.struts.actions.LookupDispatchAction.execute 7424717595 900
com.bea.medrec.actions.BaseLookupDispatchAction.execute 7534136628 1350
```

The method of interest is

```
com.bea.medrec.actions.ApprovePatientRequestAction.approve().
```

This is used to approve Patient registrations. The next step is to look at the call flow for the `ApprovePatientRequestAction.approve()` method. The `cf-gen.d` script that was explained earlier can be used for this purpose. As the requirement is to now instrument the entire application the `embedProbes.conf` file will have this single line.

```
EmbedProbes.conf
*medrec*::*
```

Since DFrame is dynamic, there is no need to restart WebLogic. Rather, we simply rerun the `embedProbes` command.
Here is the output from `cf-gen.d`

```plaintext
<table>
<thead>
<tr>
<th>TOTAL</th>
<th>COUNT</th>
<th>CLASS.METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>205810770</td>
<td>164</td>
<td>value.Mail.setSubject</td>
</tr>
<tr>
<td>252918345</td>
<td>164</td>
<td>value.Mail.setMessage</td>
</tr>
<tr>
<td>284077736</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_needsRemove</td>
</tr>
<tr>
<td>299395665</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_getIsLocal</td>
</tr>
<tr>
<td>329036802</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_setPrimaryKey</td>
</tr>
<tr>
<td>334339514</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_setM2NInsert</td>
</tr>
<tr>
<td>335827940</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_superEjbStore</td>
</tr>
<tr>
<td>340446135</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_isBeanStateValid</td>
</tr>
<tr>
<td>340885100</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS.setStatus</td>
</tr>
<tr>
<td>345715683</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_getStmtArray</td>
</tr>
<tr>
<td>345814944</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL.beanIsLoaded</td>
</tr>
<tr>
<td>351122712</td>
<td>164</td>
<td>controller.AdminSessionEJB_elen28_Impl__WL_getMethodState</td>
</tr>
<tr>
<td>352603884</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_getIsModifiedUnion</td>
</tr>
<tr>
<td>359764001</td>
<td>164</td>
<td>actions.AdminBaseLookupDispatchAction.getAdminSession</td>
</tr>
<tr>
<td>390890613</td>
<td>164</td>
<td>value.Mail.&lt;init&gt;</td>
</tr>
<tr>
<td>399268601</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_setBeanParamsForUpdateArray</td>
</tr>
<tr>
<td>406804700</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_perhapsReloadOptimisticColumn</td>
</tr>
<tr>
<td>413264278</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__ELOImpl.getPrimaryKey</td>
</tr>
<tr>
<td>429797619</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_setNonFKHolderRelationChange</td>
</tr>
<tr>
<td>432016988</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_setEntityContext</td>
</tr>
<tr>
<td>443350947</td>
<td>164</td>
<td>entities.UserEJB_mc63ls__WebLogic_CMP_RDBMS__WL_initialize_persistent</td>
</tr>
<tr>
<td>456965166</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_setOperationsComplete</td>
</tr>
<tr>
<td>536548614</td>
<td>164</td>
<td>value.Mail.setTo</td>
</tr>
<tr>
<td>584335719</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_perhapsTakeSnapshot</td>
</tr>
<tr>
<td>601060270</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_loadGroupFromRS</td>
</tr>
<tr>
<td>621359885</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_setBeanStateValid</td>
</tr>
<tr>
<td>642478249</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_superEjbLoad</td>
</tr>
<tr>
<td>633241975</td>
<td>328</td>
<td>controller.AdminSessionEJB_elen28_Impl__WL_getMethodState</td>
</tr>
<tr>
<td>662904375</td>
<td>328</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_setLoadUser</td>
</tr>
<tr>
<td>701333865</td>
<td>328</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_initialize</td>
</tr>
<tr>
<td>728054238</td>
<td>328</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_setBusy</td>
</tr>
<tr>
<td>749883071</td>
<td>328</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_getPKFromRS</td>
</tr>
<tr>
<td>843279559</td>
<td>328</td>
<td>actions.BaseLookupDispatchAction.getGetSessionAttribute</td>
</tr>
<tr>
<td>846924327</td>
<td>328</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_setMethodState</td>
</tr>
<tr>
<td>849259783</td>
<td>328</td>
<td>actions.BaseLookupDispatchAction.getMessage</td>
</tr>
<tr>
<td>990548147</td>
<td>328</td>
<td>beans.PatientBean.getEmail</td>
</tr>
<tr>
<td>996094150</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_loadGroupByIndex</td>
</tr>
<tr>
<td>1064363401</td>
<td>328</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_isBusy</td>
</tr>
<tr>
<td>1070016540</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_resetIsModifiedVars</td>
</tr>
<tr>
<td>11220263127</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_doCheckExistsOnMethod</td>
</tr>
<tr>
<td>1227214624</td>
<td>164</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_setBeanParamsForBulkUpdateArray</td>
</tr>
<tr>
<td>1282653505</td>
<td>492</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_getIsRemoved</td>
</tr>
<tr>
<td>1518162525</td>
<td>492</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_getMethodState</td>
</tr>
<tr>
<td>1518162525</td>
<td>492</td>
<td>entities.UserEJB.mc63ls__WebLogic_CMP_RDBMS__WL_getMethodState</td>
</tr>
</tbody>
</table>
```

The output above shows that ApprovePatientRequestAction.approve() method took 32.5 secs to complete. About 22 sec in that was spent executing WebLogic_CMP_RDBMS methods. That is about 67% of the time. The WebLogic_CMP_RDBMS methods are part of the Oracle WebLogic Server CMP model. Optimizing the CMP layer can provide good performance gains in the application. A complete analysis and solution to this issue is beyond the scope for this document, but it is clear that DFrame can be part of the iterative profiling and analysis required to get to the root cause of complex performance issue.
How much time is my application spending in Java EE events?

Getting a break down of components including HTTP, net, and JDBC would be very useful in any performance tuning exercise. The `dgen.d` script can be used to get a breakdown of time spent in any set of mapped probes. It is a generic script that has been developed to be widely applicable.

For this example, we will create a set of mapped probes for Java EE events using the following configuration file.

```
embedProbes.conf

*jdbc*:execute:entry:sql-start
*jdbc*:execute:return:sql-end
weblogic.servlet.internal.ServletRequestImpl:run:entry:servlet-start
weblogic.servlet.internal.ServletRequestImpl:run:return:servlet-end
weblogic.servlet.jsp.JspBase:service:entry:jsp-start
weblogic.servlet.jsp.JspBase:service:return:jsp-end
java.net.ServerSocket:accept:entry:net-start
java.net.ServerSocket:accept:return:net-end
*jms*:beforeCompletion:entry:jms-start
*jms*:afterCompletion:return:jms-end
```

Here is the output from the script for a 2 minute run using `dgen.d`:

```
# ./dgen.d

<table>
<thead>
<tr>
<th>EVENT</th>
<th>TIME SPENT(ns)</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsp</td>
<td>1065509490</td>
<td>4500</td>
</tr>
<tr>
<td>sql</td>
<td>1593455887</td>
<td>524</td>
</tr>
<tr>
<td>jms</td>
<td>2548551569</td>
<td>225</td>
</tr>
<tr>
<td>http</td>
<td>12940505251</td>
<td>4725</td>
</tr>
<tr>
<td>servlet</td>
<td>19580276202</td>
<td>5850</td>
</tr>
<tr>
<td>net</td>
<td>120245097297</td>
<td>677</td>
</tr>
</tbody>
</table>
```

Note that even though the net event looks like it is taking all of the 120 sec. This is a little surprising at first glance. It would be interesting to see what is being executed by the net events. A couple methods of drill down was already demonstrated earlier. The next section takes a look at the operating system layer to see details of what the net events were doing from the operating system point of view.

DFrame uses standard D-script syntax. This allows DFrame to tie in operating system events to events in the Oracle WebLogic Server. The next section introduces some ideas on debugging such interactions.
How much time do Java EE events spend in System Calls?

The previous sections documented how to use DFrame to observe interesting areas in the Java EE application code itself. DFrame can also be used to track down application interactions with the underlying Operating System. This can help answer the question why he “net” Java EE event takes so much time.

One simple starting point is to find out the system calls that are executed by various Java EE components. The following configuration file is used for this purpose. This file is included in DFrame an example in the configFiles directory. This config file creates mapped probes for various Java EE events:

```
embedProbes.conf

*jdbc*:execute:entry:sql-start
*jdbc*:execute:return:sql-end
weblogic.servlet.internal.ServletRequestImpl:run:entry:servlet-start
weblogic.servlet.internal.ServletRequestImpl:run:return:servlet-end
weblogic.servlet.jsp.JspBase:service:entry:jsp-start
weblogic.servlet.jsp.JspBase:service:return:jsp-end
java.net.ServerSocket:accept:entry:net-start
java.net.ServerSocket:accept:return:net-end
*jms*:beforeCompletion:entry:jms-start
*jms*:afterCompletion:return:jms-end
```

Once the embedProbes command is used to embed these probes into the running application, the D-script `whocalled syscall.d` can be used to find the system calls that are called by Java EE events. Sample output from this script is shown in the following table.

```
# .\whocalledsyscall.d
EVENT      SYSCALL          TIME SPENT  COUNT
http        getuid           6127        2
http        lwp_park         10634       2
http        setsockopt       10752       1
http        getsockname      21234       1
http        so_socket        42581       1
http        doorfs           61565       2
http        resolvepath      69866       4
jsp         lwp_cond_wait    340594      2
http        llseek           1136698     233
http        lwp_cond_signal  2571299     265
jsp         lwp_cond_wait    2696325      5
http        connect          5104904     1
jms         lwp_cond_signal  5060517      699
sql         send             9745277     532
jms         send             12132956     932
http        lwp_cond_wait    9236797     1
jsp         stat64          25937378     4222
jsp         send             46104018     4660
servlet     read             52077033     22368
servlet     stat64          75195080     7452
http        send             110539986    9786
http        stat64          133061860    17946
jsp         read             160578459    67296
jms         read             328657860    932
jms         lwp_cond_wait   632974675     673
sql         read             1843849325    532
http        read             5816698246   107274
```
The output shows that the “net” Java EE event took almost all its time executing the accept() system call. So it appears that it was waiting for requests to come in from the load generator. This looks like normal application server behavior.

Furthermore, we see a lot of `read()` system calls being executed. It may be good area to drill down further. DTrace can be used to find out the names of the files that are read by the Oracle WebLogic Server. This document does not go into the details of debugging the IO subsystem but a simple D-script that can print out all the files that are opened by Oracle WebLogic Server is included below.

```
#!/usr/sbin/dtrace -qs
syscall::read::entry
/execname=="java"/
{
    @[fds[arg0].fi_pathnam]=count();
}
```

This section demonstrated the use of DFrame to observe the Oracle WebLogic Server's interaction with the Solaris Operating System. Using similar methods DFrame can be used to observe multi-tier applications.
Quantifying the Probe Effect

One of the main goals of any monitoring system is to be as unobtrusive as possible. To reduce the overhead due to monitoring, we must be able to first calculate that overhead.

The Dynamic Tracing Framework for WebLogic Server is designed to be very efficient. A variety of performance data was collected to quantify the overhead of DFrame. First, the MedRec workload was executed with no DFrame probes to get a baseline. Then other scenarios were executed as documented in the table below.

<table>
<thead>
<tr>
<th>TRANSACTION COUNT</th>
<th>% OVERHEAD</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>No probes - base line</td>
<td>2097</td>
<td>N/A</td>
</tr>
<tr>
<td>Enabled Java EE probes</td>
<td>2061</td>
<td>1.7%</td>
</tr>
<tr>
<td>Coarse grained probes. Instrumented all the methods in all the classes of MedRec</td>
<td>2034</td>
<td>3%</td>
</tr>
<tr>
<td>Java EE probes + D-scripts</td>
<td>2043</td>
<td>2.60%</td>
</tr>
<tr>
<td>All MedRec + D-script</td>
<td>2016</td>
<td>3.8%</td>
</tr>
</tbody>
</table>

It is clear that the DFrame overhead is extremely low. Also, it is important to remember that DFrame is completely dynamic, so that overhead is incurred only during intervals when data is being collected.

System administrators can automate the process of data collection by creating D-scripts that look for certain predefined conditions. When these conditions occur another D-script that collects more
detailed information can be started. DTrace provides some excellent features to facilitate these
automation.

The “-w” option permits D-scripts to start off other D-scripts. The system() action (similar to the
system() function in C) can be used to start of D-scripts. DTrace provides the Tick provider and
the exit() action to automatically stop the execution the D-script after a predefined period.
Detailed Descriptions of Bundled D-scripts

Bundled D-scripts details
In this section, we provide a brief explanation of each of the bundled D-scripts.

**count_methods.d**

This script produces summary information for all the methods that are invoked. It counts the number of times each method is invoked. The granularity of the observation of this script can be controlled by the `embedProbes.conf` file.

For example if the `embedProbes.conf` file contains a line

```
*medrec*:*
```

Then this will count all the methods that are executed in the MedRec application.

If the `embedProbes.conf` file contains:

```
*action*:*
```

Then `count_methods.d` will count all the actions that are executed in the MedRec application.

**count_methods.d**

```bash
#!/usr/sbin/dtrace -qs
weblogic*:::method-entry
{
   @[copyinstr(arg0),copyinstr(arg1)]=count();
}
END
{
   printf("%-60s %-20s %10s\n","CLASS","METHOD","COUNT");
   printa("%-60s %-20s %@10s\n",@);
}
```

Sample Output:

```
<table>
<thead>
<tr>
<th>CLASS</th>
<th>METHOD</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.bea.medrec.actions.AdminLoginAction</td>
<td>authenticate</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.actions.AdminLogoutAction</td>
<td>executeAction</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.actions.ApprovePatientRequestAction</td>
<td>approve</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.actions.ApprovePatientRequestAction</td>
<td>composeMail</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.entities.UserEJB</td>
<td>&lt;init&gt;</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.entities.UserEJB</td>
<td>ejbCreate</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.value.Mail</td>
<td>&lt;init&gt;</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.value.Mail</td>
<td>setMessage</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.value.Mail</td>
<td>setSubject</td>
<td>227</td>
</tr>
<tr>
<td>com.bea.medrec.value.Mail</td>
<td>setTo</td>
<td>227</td>
</tr>
</tbody>
</table>
```

Note: the above output was produced with an `embedProbes.conf` file that contained the following:
This script produces a summary of the time that each of the method spends executing. Like the `count_methods.d`, the granularity of the observation of this script can be controlled by the `embedProbes.conf` file.

```
#!/usr/sbin/dtrace -qs
weblogic*:::method-entry
{
    this->cn = copyinstr(arg0);
    this->mn = copyinstr(arg1);
    self->ts[this->cn,this->mn]=timestamp;
}
weblogic*:::method-return
{
    this->cn = copyinstr(arg0);
    this->mn = copyinstr(arg1);
}
weblogic*:::method-return
//self->ts[this->cn,this->mn]/
{
    @s[this->cn,this->mn]=sum(timestamp - self->ts[this->cn,
this->mn]);
    @c[this->cn,this->mn]=count();
    self->ts[this->cn,this->mn]=0;
}
END
{
    printa("%50s.%10s  %@15d %@8d
",@s,@c);
}
```

**SAMPLE OUTPUT:**

<table>
<thead>
<tr>
<th>CLASS.METHOD</th>
<th>TIME(ns)</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.bea.medrec.value.Address.setState</td>
<td>7845840</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getState</td>
<td>7867625</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getDob</td>
<td>16689994</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.beans.UserBean.toUser</td>
<td>17364983</td>
<td>205</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getMiddleName</td>
<td>17699911</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getFirstName</td>
<td>17980862</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getLastName</td>
<td>17980862</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.setStreetName2</td>
<td>8343559</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.setStreetName1</td>
<td>8397772</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getStreetName1</td>
<td>8423503</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getStreetName2</td>
<td>8430561</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.actions.SearchResultsAction.processSearch</td>
<td>4212356200</td>
<td>205</td>
</tr>
<tr>
<td>com.bea.medrec.actions.SearchResultsAction.search</td>
<td>4261589358</td>
<td>205</td>
</tr>
<tr>
<td>com.bea.medrec.actions.PhysBaseLookupDispatchAction.execute</td>
<td>9904998643</td>
<td>1230</td>
</tr>
<tr>
<td>com.bea.medrec.filters.RequestEncodingFilter.doFilter</td>
<td>15987611900</td>
<td>4096</td>
</tr>
</tbody>
</table>
Dynamic Tracing Framework for Oracle WebLogic Server on Oracle Solaris

medrec_http_methods.d

This script lists all the methods that are used to service the HTTP request. If the embedProbes.conf enables exhaustive instrumentation like *medrec*:*, then this will list all the MedRec methods that are called to service the HTTP request. If the instrumentation is more targeted, for example *Admin*:*, then this will list the methods in the Admin class only.

MEDREC_HTTP_METHODS.D

```bash
#!/usr/sbin/dtrace -qs
weblogic*:::event
/copyinstr(arg0)="http-start"/
{
    self->in_http=1;
}
weblogic*:::method-entry
/self->in_http/
{
    this->cn = copyinstr(arg0);
    this->mn = copyinstr(arg1);
    self->ts[this->cn,this->mn]=timestamp;
}
weblogic*:::method-return
/self->in_http/
{
    this->cn = copyinstr(arg0);
    this->mn = copyinstr(arg1);
}
weblogic*:::method-return
/self->in_http && self->ts[this->cn, this->mn]/
{
    @s[this->cn, this->mn]=sum(timestamp - self->ts[this->cn, this->mn]);
    @c[this->cn, this->mn]=count();
    self->ts[this->cn, this->mn]=0;
}
END
{
    printf("%50s.%-10s  %15s %8s\n","CLASS","METHOD","TIME","COUNT");
    printa("%50s.%-10s  %@15d %@8d\n",@s,@c);
}
```

SAMPLE OUTPUT.

<table>
<thead>
<tr>
<th>CLASS_METHOD</th>
<th>TIME (ns)</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.bea.medrec.value.Patient.setPhone</td>
<td>7764217</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.setCity</td>
<td>7772288</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Patient.getSSn</td>
<td>7814521</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getEmail</td>
<td>781248</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getCity</td>
<td>7822853</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getState</td>
<td>7845840</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.value.Address.getState</td>
<td>7867625</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getDob</td>
<td>16364983</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.beans.UserBean.toUser</td>
<td>17364983</td>
<td>205</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getMiddleName</td>
<td>17699911</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.beans.BaseBean.getId</td>
<td>17783028</td>
<td>2665</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getFirstName</td>
<td>18089213</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getLastName</td>
<td>17980862</td>
<td>2460</td>
</tr>
<tr>
<td>com.bea.medrec.beans.Address.setStreetName2</td>
<td>8343559</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.beans.Address.getStreetName</td>
<td>8397772</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.beans.Address.getStreetName</td>
<td>8423503</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.beans.PatientBean.getMiddleName</td>
<td>8430561</td>
<td>1228</td>
</tr>
<tr>
<td>com.bea.medrec.actions.SearchResultsAction.processSearch</td>
<td>4212356200</td>
<td>205</td>
</tr>
<tr>
<td>com.bea.medrec.actions.SearchResultsAction.search</td>
<td>4261589358</td>
<td>205</td>
</tr>
<tr>
<td>com.bea.medrec.actions.PhysBaseLookupDispatchAction.execute</td>
<td>4277869374</td>
<td>205</td>
</tr>
<tr>
<td>com.bea.medrec.actions.BaseAction.execute</td>
<td>7019780498</td>
<td>3485</td>
</tr>
</tbody>
</table>
Dynamic Tracing Framework for Oracle WebLogic Server on Oracle Solaris

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Method Name</th>
<th>Code Address</th>
<th>CPU Time</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.bea.medrec.actions.BaseLookupDispatchAction.execute</td>
<td>9904998643</td>
<td>1230</td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.bea.medrec.filters.RequestEncodingFilter.doFilter</td>
<td>15987611900</td>
<td>4096</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the above result was from a 2 minute run with the embedProbes.conf file that contains
*medrec*: *
This script takes a name of a class.method as its only input and prints the call flow for that method and the time spent in each of the called methods.

```plaintext
#pragma D option strsize=204800
#pragma D option dynvarsize=30m
#pragma D option switchrate=1000hz
#pragma D option aggsize=40m
#pragma D option aggrate=100hz

weblogic*:::method-entry
{this->cm=strjoin(strjoin(copyinstr(arg0),"."),copyinstr(arg1));}

weblogic*:::method-entry
/strstr(this->cm,$$1)!=0/
{self->ts=timestamp;
 self->cf=strjoin("->", this->cm);
 self->ats[this->cm]=timestamp;
}

weblogic*:::method-entry
/self->ts && strstr(this->cm,$$1)==0/
{self->cf=strjoin(self->cf,strjoin("\n->",this->cm));
 self->ats[this->cm]=timestamp;
}

weblogic*:::method-return
/self->ts/
{this->cm=strjoin(strjoin(copyinstr(arg0),"."),copyinstr(arg1));}

weblogic*:::method-return
/self->ts && strstr(this->cm,$$1)==0/
{self->cf=strjoin(self->cf,strjoin("\n<-",this->cm));
 @cc[this->cm]=count();
 @ct[this->cm]=sum(timestamp - self->ats[this->cm]);
 self->ats[this->cm]=0;
}

weblogic*:::method-return
/self->ts && strstr(this->cm, $$1)!=0/
{self->cf=strjoin(self->cf,"\n<-"), this->cm);
 @cc[this->cm]=count();
 @ct[this->cm]=sum(timestamp - self->ats[this->cm]);
 @cc[this->cm]=min(timestamp - self->ats[this->cm]);
 @av[this->cm]=avg(timestamp - self->ats[this->cm]);
 @ai[this->cm]=min(timestamp - self->ats[this->cm]);
 @aa[this->cm]=max(timestamp - self->ats[this->cm]);
 self->ats[this->cm]=0;
}

END
{
 printa("%s
 Total Time: %@d :: Fastest: %@d :: Slowest: %@d :: Average: %@d :: Count: %@d
",@t, @i, @a, @v, @c);
}
```
SAMPLE OUTPUT

# ./cf-gen.d Login
->com.bea.medrec.actions.AdminLoginAction.authenticate
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.BaseBean.<init>
->com.bea.medrec.beans.AdminBean.<init>
<com.bea.medrec.beans.AdminBean.<init>
->com.bea.medrec.actions.BaseAction.getRedirectPage
<com.bea.medrec.utils.MedRecWebAppUtils.isNotEmpty
<com.bea.medrec.actions.BaseAction.getRedirectPage
<com.bea.medrec.actions.AdminLoginAction.authenticate
Total Time: 506675143 :: Fastest: 11285901 :: Slowest: 21683922 :: Average: 14074309 :: Count: 36

->com.bea.medrec.actions.LoginAction.executeAction
->com.bea.medrec.actions.BaseAction.setupLocale
<com.bea.medrec.utils.MedRecWebAppUtils.getLocaleFromCookie
<com.bea.medrec.utils.MedRecWebAppUtils.isValidLocale
<com.bea.medrec.actions.BaseAction.setupLocale
<com.bea.medrec.actions.BaseAction.getMessage
<com.bea.medrec.utils.MedRecWebAppUtils.isNotEmpty
<com.bea.medrec.actions.PhysLoginAction.executeAction
Total Time: 562070338 :: Fastest: 10336476 :: Slowest: 20600211 :: Average: 15613064 :: Count: 36

->com.bea.medrec.actions.PhysLoginAction.executeAction
->com.bea.medrec.actions.BaseAction.setupLocale
<com.bea.medrec.utils.MedRecWebAppUtils.getLocaleFromCookie
<com.bea.medrec.utils.MedRecWebAppUtils.isValidLocale
<com.bea.medrec.actions.BaseAction.setupLocale
<com.bea.medrec.actions.BaseAction.getMessage
<com.bea.medrec.utils.MedRecWebAppUtils.isNotEmpty
<com.bea.medrec.actions.PhysLoginAction.executeAction
Total Time: 588936385 :: Fastest: 15259640 :: Slowest: 20723465 :: Average: 16359344 :: Count: 36

->com.bea.medrec.actions.AdminLoginAction.authenticate
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getPassword
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getPassword
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
<com.bea.medrec.beans.UserBean.getUsername
Dynamic Tracing Framework for Oracle WebLogic Server on Oracle Solaris

```java
->com.bea.medrec.beans.BaseBean.<init>
<com.bea.medrec.beans.BaseBean.<init>
->com.bea.medrec.beans.PhysicianBean.<init>
<com.bea.medrec.beans.PhysicianBean.<init>
->com.bea.medrec.actions.BaseAction.getRedirectPage
<com.bea.medrec.actions.PhysLoginAction.authenticate
<com.bea.medrec.utils.MedRecWebAppUtils.isNotEmpty
<com.bea.medrec.actions.BaseAction.getRedirectPage
<com.bea.medrec.actions.PhysLoginAction.authenticate
<com.bea.medrec.beans.UserBean.getPassword
```

**SUMMARY of all methods called from Login**

<table>
<thead>
<tr>
<th>COUNT</th>
<th>TOTAL</th>
<th>CLASS.METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>72893083</td>
<td>com.bea.medrec.value.Patient.getSSn</td>
</tr>
<tr>
<td>35</td>
<td>73205131</td>
<td>com.bea.medrec.value.Address.getCountry</td>
</tr>
<tr>
<td>35</td>
<td>87782367</td>
<td>com.bea.medrec.value.Patient.getMiddleName</td>
</tr>
<tr>
<td>35</td>
<td>95979836</td>
<td>com.bea.medrec.value.Patient.getLastName</td>
</tr>
<tr>
<td>35</td>
<td>96729579</td>
<td>com.bea.medrec.value.Patient.getEmail</td>
</tr>
<tr>
<td>35</td>
<td>9707591</td>
<td>com.bea.medrec.value.Patient.getAddress</td>
</tr>
<tr>
<td>35</td>
<td>107019505</td>
<td>com.bea.medrec.value.Address.getZipCode</td>
</tr>
<tr>
<td>35</td>
<td>110377893</td>
<td>com.bea.medrec.value.Address.getStreetName1</td>
</tr>
<tr>
<td>35</td>
<td>115508167</td>
<td>com.bea.medrec.value.Patient.getGender</td>
</tr>
<tr>
<td>35</td>
<td>115942430</td>
<td>com.bea.medrec.value.Patient.getPhone</td>
</tr>
<tr>
<td>35</td>
<td>115960361</td>
<td>com.bea.medrec.value.Address.getState</td>
</tr>
<tr>
<td>35</td>
<td>116940300</td>
<td>com.bea.medrec.value.Address.getStreetName2</td>
</tr>
<tr>
<td>35</td>
<td>148277561</td>
<td>com.bea.medrec.value.Patient.getFirstName</td>
</tr>
<tr>
<td>35</td>
<td>3581745060</td>
<td>com.bea.medrec.beans.PatientBean.&lt;init&gt;</td>
</tr>
<tr>
<td>35</td>
<td>19470932895</td>
<td>com.bea.medrec.actions.LoginAction.authenticate</td>
</tr>
<tr>
<td>36</td>
<td>25812584</td>
<td>com.bea.medrec.beans.UserBean.getPassword</td>
</tr>
</tbody>
</table>

Total Time: 731670618 :: Fastest: 12288965 :: Slowest: 28987550 :: Average: 20324183 :: Count: 36
eventflow.d

This script is similar to cf-gen.d. This script follows the call flow of Java EE events. The events are defined in the config file as mapped probes. Here is an example of the embedProbes.conf file used.

```
*jdbc*:execute:entry:sql-start
*jdbc*:execute:return:sql-end
weblogic.servlet.internal.ServletRequestImpl:run:entry:servlet-start
weblogic.servlet.internal.ServletRequestImpl:run:return:servlet-end
weblogic.servlet.jsp.JspBase:service:entry:jsp-start
weblogic.servlet.jsp.JspBase:service:return:jsp-end
java.net.ServerSocket:accept:entry:net-start
java.net.ServerSocket:accept:return:net-end
*jms*:beforeCompletion:entry:jms-start
*jms*:afterCompletion:return:jms-end
*Action:*  
```

Here is the `eventflow.d` script:

```
#!/usr/sbin/dtrace -qs
#pragma D option strsize=204800
#pragma D option dynvarsize=50m
#pragma D option switchrate=1000hz
#pragma D option aggsize=40m
#pragma D option aggrate=100hz

weblogic*:::event /copyinstr(arg0)==strjoin($$1,"-start")/
{  
    self->cf=strjoin("->", copyinstr(arg1));
    self->ts=timestamp;
}

weblogic*:::method-entry /self->ts/
{  
    this->cm=strjoin(copyinstr(arg0),".",copyinstr(arg1));
    self->cf=strjoin(self->cf,strjoin("n->", this->cm));
    self->ats[this->cm]=timestamp;
}

weblogic*:::method-return /self->ts/
{  
    this->cm=strjoin(copyinstr(arg0),".",copyinstr(arg1));
}

weblogic*:::method-return /self->ts/
{  
    self->cf=strjoin(self->cf,strjoin("\n<-",this->cm));
    @ac[this->cm]=count();
    @at[this->cm]=sum(timestamp - self->ats[this->cm]);
    self->ats[this->cm]=0;
}

weblogic*:::event /self->ts & & copyinstr(arg0)==strjoin($$1,"-end")/
{  
    this->cm=copyinstr(arg1);
    self->cf=strjoin(self->cf,"\n<-",this->cm);
    @t[self->cf]=sum(timestamp - self->ts);
    @c[self->cf]=count();
}
```
Sample output

The output depends on the definition of the mapped probes in the embedProbes.conf file. Here is the output when using the following embProbes.conf file for a few seconds of the run to observe the event flow of a JSP.

```
# ./eventflow.d jsp
^C
  --weblogic.servlet.jsp.JspBase.service
  Total Time: 91730900 :: Count: 265
  --weblogic.servlet.jsp.JspBase.service
  --org.apache.struts.config.ActionConfig.getName
    --org.apache.struts.config.ActionConfig.getScope
    --org.apache.struts.config.ActionConfig.getAttribute
    --org.apache.struts.config.ActionConfig.getName
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --org.apache.struts.action.ActionMessages.<init>
    --weblogic.servlet.jsp.JspBase.service
    Total Time: 1077598879 :: Count: 49
```

```
SUMMARY of all methods called from jsp

<table>
<thead>
<tr>
<th>TOTAL</th>
<th>COUNT</th>
<th>METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>141345869</td>
<td>385</td>
<td>org.apache.struts.action.ActionMessages.get</td>
</tr>
<tr>
<td>143399641</td>
<td>66</td>
<td>org.apache.struts.config.ActionConfig.getScope</td>
</tr>
<tr>
<td>175420766</td>
<td>66</td>
<td>org.apache.struts.config.ActionConfig.getAttribute</td>
</tr>
<tr>
<td>311064724</td>
<td>66</td>
<td>org.apache.struts.config.ActionConfig.getName</td>
</tr>
<tr>
<td>447400969</td>
<td>385</td>
<td>org.apache.struts.action.ActionMessages.&lt;init&gt;</td>
</tr>
<tr>
<td>848726224</td>
<td>165</td>
<td>weblogic.servlet.internal.StubSecurityHelper$ServletServiceAction.&lt;init&gt;</td>
</tr>
</tbody>
</table>
```
This script provides a breakdown of the time spent in mapped probes. This script assumes that each event mapped has a start point and an end point. The names of the start point of the probe will contain “-start” and the name of the end point will contain “-end”.

Here is the script.

```bash
#!/usr/sbin/dtrace -qs
weblogic*:::event
/strstr(copyinstr(arg0),"start")!=0/
{
    self->ts[stringof(strtok(copyinstr(arg0),"-")]=timestamp;
}
weblogic*:::event
/self->ts[stringof(strtok(copyinstr(arg0),"-"))]!=0 & & strstr(copyinstr(arg0),"end")!=0/
{
    this->name=copyinstr(arg0);
    this->pname=stringof(strtok(this->name,"-"));
    @ts[this->pname]=sum(timestamp - self->ts[this->pname]);
    @ct[this->pname]=count();
    self->ts[this->pname]=0;
}
END
{
    printf("%-30s %-20s %-10s
","EVENT","TIME SPENT(ns)","COUNT"),
    printa("%-30s %@-20d %@-10d
",@ts,@ct);
}
```

The output of the script will vary based on the `embedProbes.conf` file. We will use the following for this example.

```bash
*jdbc*:execute:entry:sql-start
*jdbc*:execute:return:sql-end
weblogic.servlet.internal.ServletRequestImpl:run:entry:servlet-start
weblogic.servlet.internal.ServletRequestImpl:run:return:servlet-end
weblogic.servlet.jsp.JspBase:service:entry:jsp-start
weblogic.servlet.jsp.JspBase:service:return:jsp-end
java.net.ServerSocket:accept:entry:net-start
java.net.ServerSocket:accept:return:net-end
*jms*:beforeCompletion:entry:jms-start
*jms*:afterCompletion:return:jms-end
```

Here is a sample output:

<table>
<thead>
<tr>
<th>EVENT</th>
<th>TIME SPENT(ns)</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsp</td>
<td>956005643</td>
<td>4480</td>
</tr>
<tr>
<td>sql</td>
<td>1702454758</td>
<td>1122</td>
</tr>
<tr>
<td>jms</td>
<td>2474206382</td>
<td>224</td>
</tr>
<tr>
<td>http</td>
<td>11920959396</td>
<td>4704</td>
</tr>
<tr>
<td>servlet</td>
<td>17993937858</td>
<td>5824</td>
</tr>
<tr>
<td>net</td>
<td>119689740023</td>
<td>674</td>
</tr>
</tbody>
</table>
whocalledsyscall.d

This script prints the system calls that are executed from mapped probes.

```c
weblogic*:::event
/strstr(copyinstr(arg0),"start")!=0/
{
    self->ts[stringof(strtok(copyinstr(arg0),"-"))]=timestamp;
    self->in=stringof(strtok(copyinstr(arg0),"-"));
    self->j2ee=1;
}
syscall:::entry
/self->j2ee/
{
    @c[self->in,probefunc]=count();
    self->ts[probefunc]=timestamp;
}
syscall:::return
/self->j2ee && self->ts[probefunc]/
{
    @t[self->in,probefunc]=sum(timestamp - self->ts[probefunc]);
    self->ts[probefunc]=0;
}
weblogic*:::event
/self->ts[stringof(strtok(copyinstr(arg0),"-"))]!=0 && strstr(copyinstr(arg0),"end")!=0/
{
    self->j2ee=0;
    self->in=0;
}
END
{
    printf("%-10s %-18s %-15s %-10s
","EVENT","SYSCALL","TIME SPENT","COUNT"; 
    printa("%-10s %-18s %@-15d %@-10d
",@t,@c);
}
```

Here is the `embedProbes.conf` file used:

```bash
*jdbc*:execute:entry:sql-start
*jdbc*:execute:return:sql-end
weblogic.servlet.internal.ServletRequestImpl:run:entry:servlet-start
weblogic.servlet.internal.ServletRequestImpl:run:return:servlet-end
weblogic.servlet.jsp.JspBase:service:entry:jsp-start
weblogic.servlet.jsp.JspBase:service:return:jsp-end
java.net.ServerSocket:accept:entry:net-start
java.net.ServerSocket:accept:return:net-end
*jms*:beforeCompletion:entry:jms-start
*jms*:afterCompletion:return:jms-end
```

Here is the sample output:

```bash
% ./dgen.d
```

<table>
<thead>
<tr>
<th>EVENT</th>
<th>TIME SPENT(ns)</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsp</td>
<td>9560055643</td>
<td>4480</td>
</tr>
<tr>
<td>sql</td>
<td>1702454758</td>
<td>1122</td>
</tr>
<tr>
<td>jms</td>
<td>2474206392</td>
<td>224</td>
</tr>
<tr>
<td>http</td>
<td>11920959396</td>
<td>4704</td>
</tr>
<tr>
<td>servlet</td>
<td>17993937858</td>
<td>5824</td>
</tr>
<tr>
<td>net</td>
<td>119689740023</td>
<td>674</td>
</tr>
</tbody>
</table>

---

Dynamic Tracing Framework for Oracle WebLogic Server on Oracle Solaris

ORACLE®
Resources

DTrace Resources
More details about DTrace can be found in the following locations:

   General collection of DTrace information

   DTrace community page on Opensolaris.org site
   http://hub.opensolaris.org/bin/view/Community+Group+dtrace/WebHome

   DTrace hands on Lab
   http://dtracehol.com/

   DTrace Toolkit
   http://hub.opensolaris.org/bin/view/Community+Group+dtrace/dtracetoolkit

BTrace Resources

   The BTrace project page
   http://projectkenai.com/projects/btrace/pages/Home

   The BTrace Users Guide

Pre-release access to Dframe

Please email dframe-interest_us@oracle.com if you would like to preview a pre-release version of DFrame