An Oracle White Paper
March 2011

Techniques for Testing Performance/Scalability and Stress-Testing ADF Applications

Introduction

Before putting any application with a significant user community into production, it's essential to conduct performance/scalability testing as well as stress testing. These activities help ensure that the application is suitably responsive and stays up while in production use, and that the software and hardware configuration needed to provide satisfactory results for the users is in place.

Performance/scalability and stress testing shouldn't be a one-time activity done shortly before the application is expected to go live. Rather, testing should be ongoing throughout the entire application lifecycle, starting in very early phases of development – perhaps even with the proof of concept – and continuing on through maintenance releases. By starting early and making it an ongoing activity, you can get alerted early on to possible design problems, and quickly spot regressions introduced by enhancements and bug fixes. If you wait until the application is almost ready to go before you start testing, you are likely to encounter unpleasant surprises that may result in a substantially delayed go live date. Establishing a testing regimen early can also help avoid wasting time on optimizing code that has no significant impact on performance.

While it is fairly easy to measure the performance of a Java API invoked from a unit test, it gets considerably more challenging to measure the scalability and performance of an interactive, multi-user application. Doing so requires special tools and techniques, and a successful measurement and result is highly dependent upon a realistic assessment of the number of users and how they will use the application.

This paper examines some tools and techniques for testing the scalability of ADF Faces applications, based on Oracle internal experience.

Performance/Scalability Testing

Multiple factors affect user satisfaction with an application, chief among them the user interface and how well suited it is to the activities it is designed to support. Even the most appealing user interface, however, cannot guarantee user satisfaction if the perceived responsiveness of the application is poor. If users feel application sluggishness is interfering with their productivity, they will be frustrated and unhappy.

The critical measure of load testing tools is *response time*, defined as the total time from the first byte of the request sent by the client (browser) to the last byte received to complete any request/response interaction. This includes the http request for HTML content, as well as any additional requests for images and other resources, and includes network latency as well as server elapsed time.

It doesn't take any special tools or techniques to assess whether an application can perform adequately for a small user community. This can be done subjectively by having a team of people testing the application manually. The mission of performance and scalability testing is to assess whether the application with a sizable user community can scale to the expected number of users without having to assemble and subject that many people to these experiments. It's also important to judge whether the goal is within reach while the application is being developed, so that necessary performance related changes can be made to the application or the hosting environment configuration in a planned, methodical way.

Establishing the Goal

A performance/scalability goal for an application can usually be simply stated. For example:

> *The application must accommodate for long periods of time 300 simultaneously active users with response time average not to exceed 2 seconds. The application must accommodate a peak load of 600 active users for short durations of time with response time average not to exceed 7 seconds.*

Notably missing is a goal for maximum response time; the average alone is usually sufficient to ensure good results.

It's easily possible to create a much more complex goal citing response times for particular activities, but this is difficult to manage in a scalability test setting, and doesn't meaningfully improve the outcome. In any case, how well the test results translate into user satisfaction after the application goes live depends mostly upon how realistic the user workload simulation is during testing.

Approach

The approach is simple: create a simulated user workload, or, a *click stream*, that adequately represents how the users will commonly use the application, and apply this load in a repeatable and scalable way to the application deployed in a similar configuration to the target deployment environment. The tricky part is crafting a simulated workload that can sufficiently model how real users will use the site, and integrate this into the development process. Challenges may include:

- Some of the application features may not be available at the time the click stream is first created
- The most common activities may not be entirely clear in early stages of development
- If the simulated workload changes over time to adapt to evolving features, UI, or better understanding of how users will work with the application, comparisons of results across time won't provide information about regressions. (However, these discontinuities, if done infrequently, can be manageable.)
- If the workload is highly complex and the UI keeps evolving, it becomes a chore to maintain the click stream

One useful technique is to identify a small set of end-to-end user scenarios that represent common and important tasks. "End-to-end" means starting with the user logging in, navigating through the interactions they need to perform the task, and logging out.

If you end up with, say, 10 of these task-oriented scenarios, you can create a representative workload by combining them into a composite click stream that represents what a simulated user might do in the real world. For example, simulated user "Take Email Orders" performs task "Create a New Order" three times in fairly quick succession, waits awhile, performs task "Check on Order Status", waits awhile, performs task "Create a New Order" again, waits a bit more, then executes task "Update Shipping Instructions", and so on. Simulated user "Customer Support" performs task "Change Customer Address", waits for a bit, then submits "Cancel Order", etc.

After you develop click streams for different simulated types of users, you can determine the relative ratios for each user type. Let's say the relative ratios go like this:

- Three of "Shipping Clerk"
- Two of "Take Email Order"
- One of "Take Phone Order"
- One of "Customer Support"
- Two of "Manager"
- One of "Vice President"

The next step is to package these up into a load driver that executes them all at once. Then it becomes easy to scale your target number of concurrent users by copying and firing up more drivers; for example, executing 10 drivers means 100 users, 50 drivers means 500 users, and so on. As you are unlikely to come near your goal at the beginning, you can scale up gradually until you hit the "wall": the point when a server becomes overloaded, signified by a dramatic drop-off in performance. As your tuning of the application improves, you can continue to scale up. Note for better realism, the start time for each driver should be offset slightly, using a predefined offset schedule.

Think Time

Once you have defined the task scenarios, the next step is to record the click stream. There are a number of tools to help you with this, discussed later in this paper. An important thing to get right is "think time". In real life, as humans interact with an application, they need time to process just presented information before they can act on it. If it is an application they use infrequently, they also need time to study the UI before they can determine how to take their next action, and this is significant. On the flip side, if it is an application they use frequently, users become very skilled with it and need very little time to study the UI.

It's very easy to get the pacing wrong for think time and dramatically skew the test results in either direction. One way to help counteract this is to get representative users to run through the task scenarios and record their results in order to help you develop more realistic think times.

Parameterize the Task Scripts

Most load execution tools give you a way to parameterize click stream scripts; for example, you can parameterize which user to log in as, which order number to use, and so on. Some tools provide support for randomized values to be automatically substituted. Using these features will make your task streams more reusable.

Iteration Pacing

As you create the composite click stream, simulating a type of user by concatenating together task-oriented streams, it's important to model some delay time between each task (iteration). This reflects what happens in the real world, in which a human must contend with other activities besides just interacting with the application. Between tasks they may need to talk to a colleague or boss, grab a cup of coffee, use a different application, eat lunch, and so on. Just as with think time, it's easy to get the pacing wrong and dramatically skew the test results in either direction. One technique is to observe real users in settings where they aren't very aware they are being observed, and get some real world timing data to help guide you.

New Session vs. Shared Session

We typically find it convenient to model each task click stream as a whole sequence end-to-end, from logging in to logging out. However, if the tasks are quite short and expected to be repeated in the same session, you'll want to model the task sequences differently. For example, you can create login and logout task sequences as individual task streams, then insert them in appropriate places into the composite task stream for a simulated user type. For example, the "Take Email Orders" composite click stream could be modeled as "login", "create order 1", "create order 2", "create order 3", "logout", "login", "check on order status", etc.

Number of Simulated Users

It's easy to be overly optimistic about the estimated active and peak users the application will be supporting. It's better to estimate too high than too low, but if the estimated user base is excessively unrealistic, it creates too much pressure on the design and tuning of the application, and could result in allocating too much hardware.

Because a new application often replaces an existing application, this can be the best place to gather realistic data on active and peak usage. Even a brand new application may have the same user community as an existing application, which can serve as a source of informed usage data.

Session Timeout

It is extremely important to set session timeout appropriately for the application for a variety of reasons, including application security. Many users of web applications forget to explicitly logout. Even if they do remember, when they logout a new, "anonymous" user session is created which is left running in case they log in as someone else (at that point the anonymous session gets converted to a real session). The state maintained for each session in an ADF application is significant, even for anonymous sessions. Session state remains in memory until the timeout expires.

Session timeout should be set as low as can be tolerated. A good rule of thumb is to set the session timeout so low that users of the application complain, and gradually add a little more time until the volume of complaints tapers off.

If the application is expected to support a large number of users, such as a public facing website, it is essential that timeout be set extremely low (less than five minutes). If state needs to be preserved longer than that, other techniques should be used to achieve this.

If you don't remember to set the timeout, the default is 45 minutes. If your click stream sessions are actually only a few minutes long and you haven't explicitly set the session timeout, during scalability testing you will end up consuming extreme amounts of memory, often enough to exceed the physical memory of the host machine. The difference in memory consumption over time, starting from the beginning of the test, looks like this:

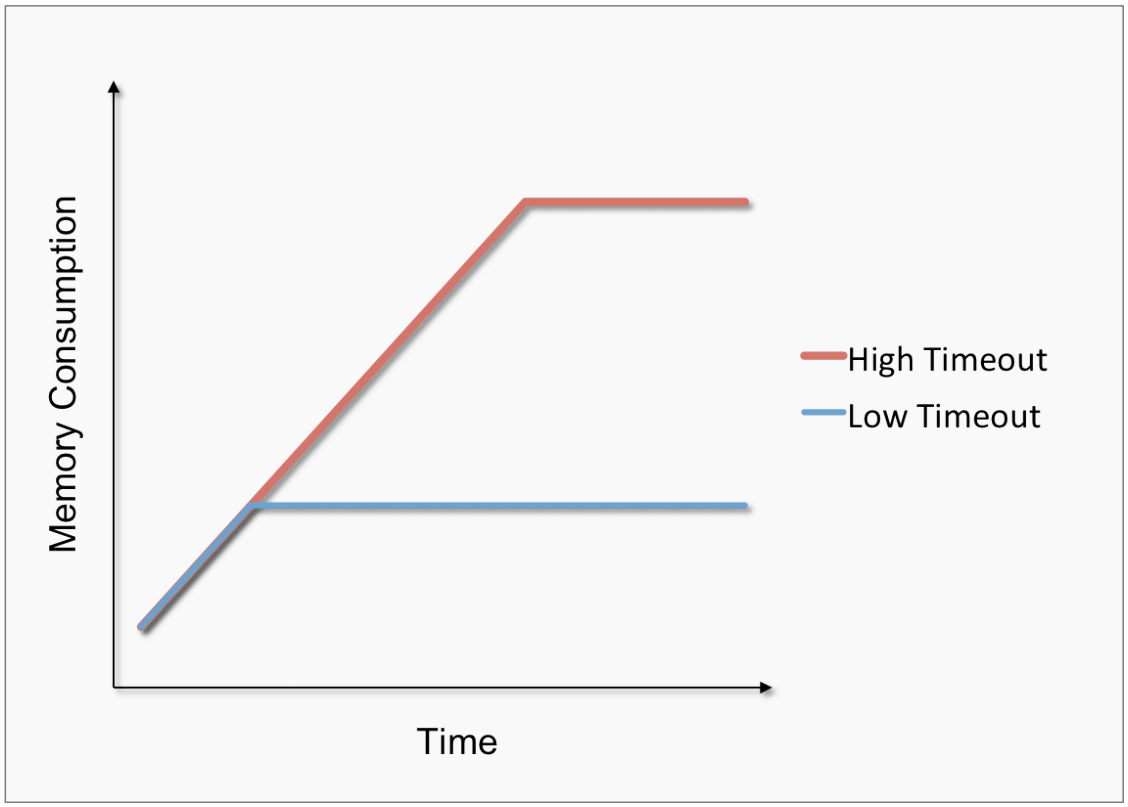

Figure 1 – Graph of Total Memory Consumption Given Different Session Timeouts

While both curves eventually flatten out (assuming no memory leaks), a high timeout will result in large amounts of memory being allocated that belong to defunct sessions waiting to expire.

To set the session timeout to five minutes, add or modify the following to the application's web.xml file:

```
<session-config>
   <session-timeout>
      5
   </session-timeout>
</session-config>
```

Preserving View State, Controller State, and Other ADF Query Parameters

ADF Faces is stateful and maintains state in two places:

- View state is preserved in a hidden form field on each page. This hidden form field can either contain an ID that is used to locate its view state on the server (server state saving), or it will contain a hexadecimal encoding of the view state (client state saving)
- Controller state and other session info is preserved as query parameters on the URL (e.g., _adf.ctrl-state, _afrLoop, _afrWindowMode, _afrWindowId, _afPfm, _rtrnId)

The state that gets recorded while making the click stream won't be correct when running tests later, as this will be information for a long-expired session. Instead, this state has to be created fresh and updated for the sessions that are created as the test is running. By default load test tools do not update this state, but it is possible to code/script a solution to preserve it. This process, often called "correlation", is critically important. Without it, the application will not behave correctly when running the simulated user load, and will leak memory that otherwise wouldn't happen.

See Appendices A and B for examples of how to implement correlation using Oracle Application Testing Suite and HP Loadrunner.

Running Tests Using Less Hardware

While developing an application that is expected to host a very large number of users, sometimes the target host equipment isn't available. Perhaps the equipment won't be ordered until it is better understood what exactly is required to support the application. Or the equipment is expensive and the order won't be placed until the project is farther along. In any case, you can use smaller equipment on hand and make rough estimates as to what larger equipment can handle. If the web application is designed and implemented without many intrinsic bottlenecks, CPU cores and memory scale fairly linearly. That is, twice as many cores and twice as much main memory should accommodate roughly twice as many users.

Sometimes you are limited in how many drivers (simulated users) you can run because of the load testing software license costs, or the equipment needed to host all the drivers. If you are limited in how many drivers you can run, you can run a smaller number of simulated users with reduced think times. Suppose you assume N active users with T average think time (which in this calculation includes pacing intervals) and you have an average interaction response time of R. The average number of interactions per second is calculated as:

$$I = N / (T + R)$$

Using an example of 300 users with an average of 28 seconds of think time, and an average of 2 seconds response time, then the number of interactions per second is 300/30, which is 10 interactions per second. You can approximate the same load with fewer simulated users by adjusting think time using the following formula:

$$T = (N / I) - R$$

For the example above, using N of 100 users and using 10 for I as computed above, you have T=(100/10) - 2, so you achieve approximately the same workload on the web server with 100 simulated users instead of 300 users by reducing average think time from 28 to 8 seconds.

Note that this approach doesn't result in the exact same load. It may not provoke concurrency issues as the number of virtual users has shrunk, and isn't as accurate for estimating memory consumption.

While both these techniques can provide useful guidelines for how the application will perform using less test equipment, it is always more accurate to run the test on the actual target hardware with a full simulated user workload.

Sanity Checks

A good thing to do before formally starting the load testing is to run the script for a single virtual user for a few hours, and see whether the interaction response times are reasonable and whether the memory consumption levels off. If you don't get reasonable response times, or the memory continues to grow, there's something fundamentally wrong and there's no point in testing with lots of simulated users. Check and correct configuration and application design problems before proceeding.

Another good check is to run a single virtual user test as above without any think time at all. This places more stress on the application than manual testing, which can quickly expose issues that must be resolved before proceeding.

After all virtual user scripts are working well with only one simulated user at a time, next try a modest number of multiple users of different types, say 10 or so, and see if the response times and memory consumption are reasonable. This may trigger concurrency or race condition problems that haven't been seen before. Correct any gross problems before continuing.

As mentioned earlier, if the think time or iteration pacing are unrealistic in the click stream, it can dramatically skew results in either direction. Once you've run the test stream as suggested above for, say, 10 simulated users, your load testing software can tell you how many request/response interactions you are making per second. Multiply by the expected total number of users, and see if this is in the realm of what the system was expected to handle in interactions per second. If it seems much higher or lower than expected, it's quite possible you're using an invalid think time or pacing calculation in your simulated user load.

Available Tools

There are a number of tools available to create and drive a simulated user load, and to monitor the results. Such tools include:

- Oracle Application Testing Suite
- HP LoadRunner
- Segue SilkPerformer
- Rational TestStudio
- Compuware QALoad
- The Grinder (Open Source)
- Apache JMeter (Open Source)

Each of these products has strengths and weaknesses that can factor into which tool you choose, but each should be powerful enough to handle the job of creating a workload on an ADF Faces application. Some tools require more setup than others to make them effective. This paper does not compare or recommend a particular tool, as there are many product reviews, comparisons, and tutorials available on the Internet to help you choose.

Some automated testing software tools can be repurposed as load testing tools, for example, Selenium (Open Source). While these may be able to handle the job, they were designed for a different purpose and often have constraints or limits that make them less appropriate.

Oracle Application Testing Suite comes with an extension that is capable of preserving the ADF view state, controller state, and other query parameters; details can be found in Appendix A. . The other tools can be configured or programmed to handle this. Appendix B gives guidance on how to accomplish this with HP LoadRunner.

Stress Testing

In this paper we use the term *stress testing* to mean putting the application under extreme load for a long period of time. We want to place the application under higher load than it's expected to encounter in the real world to see what breaks down when pushed to the edge. This provides clues on where the application may not be as robust as it should be, and what users may see under some unusual conditions. Running a severe load for several days makes it possible to more quickly identify problems such as memory, file handle, connection, or any other kind of resource leaks, which can cause premature application failures.

Establishing the Goal

A stress test goal can usually be simply stated. For example:

> *The application should be able to run under severe load for 7 days with no serious errors and tolerable levels of growth in memory and other resources.*

Note that the goal realistically needs to assume some resource leakage, particularly memory. If an application can run for seven days under severe load without running out of memory, it's quite likely to be able to run for many weeks under normal use.

Approach

Once you've created the click streams for performance/scalability testing, the job is simple because they can be reused for stress testing. The only trick is to get the load correct.

During performance/scalability testing, you routinely push the virtual user load until you hit the "wall" – the point where the web application's response suddenly drops off non-linearly as the server becomes saturated and overloaded. The objective of stress testing is to keep the application near the edge of the "wall" for days at a time such that load-related issues such as concurrency problems are exposed, and resource leaks appear more quickly than they would under normal conditions. However, you don't want to overload the application to the point where failure is guaranteed because it never has a chance to recover from continuous overload conditions.

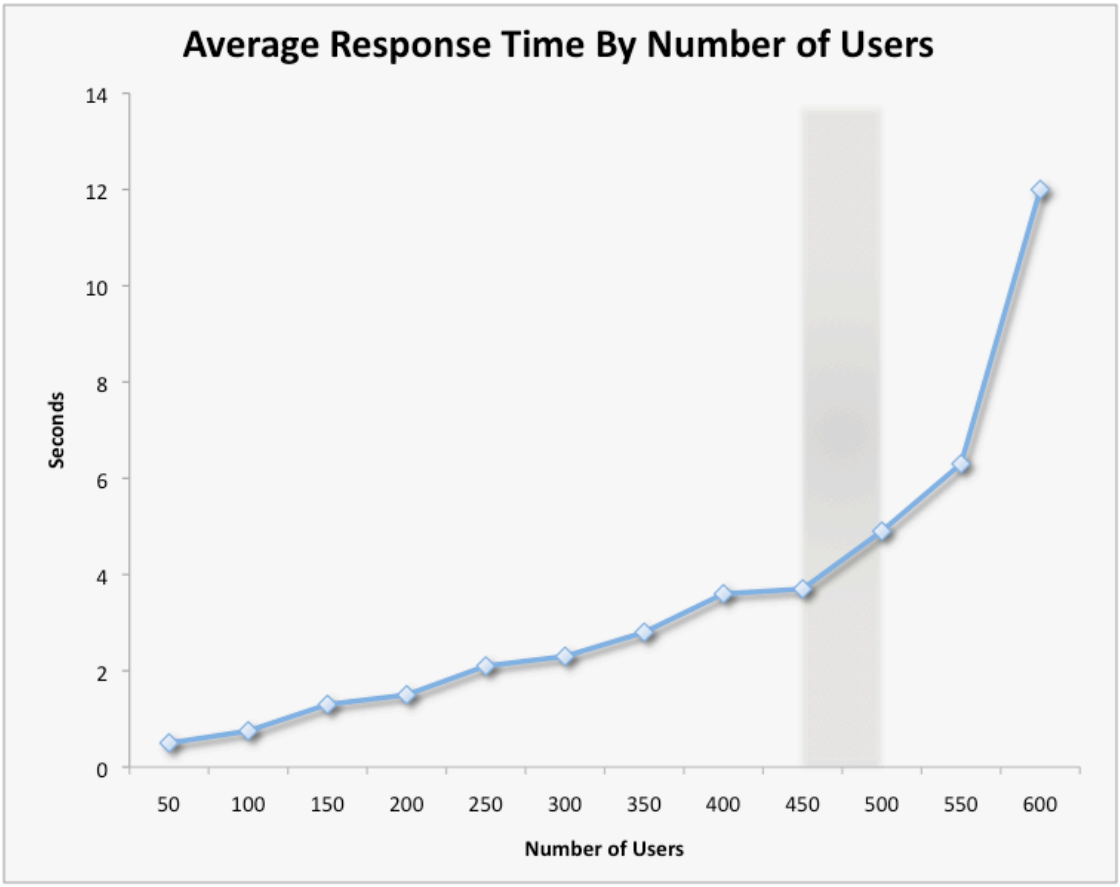

Figure 2 – Graph Showing Preferred Load Area for Stress Testing Relative to the "Wall"

The location of the "wall" moves as the application's performance is improved, bugs are fixed, and new functionality is added, or if it is hosted on equipment with differing capabilities. With each new stress test, the first step is to re-establish the location of the "wall" by scaling up/down the number of virtual users (or by adjusting the think time) and determining the point where the server becomes saturated and response times start a sudden drop off. Back off the load so that it is higher than normal, but not so high that the system never has a chance to recover from overload (Figure 2 gives some visual guidance). Let the load run for days. If there are signs in the log files of being unable to recover from overload within the first few hours, back the load down a bit further and start again.

What and How to Monitor

Integrate Performance Monitoring Into the Design

During application development it's very desirable to instrument the performance of commonly used or critical operations. For multi-user web applications, the performance instrumentation requirements

are complex, as it's very likely you'll want to be able to analyze performance for a particular request, a particular user, or when a particular set of conditions is met.

Oracle DMS (Dynamic Monitoring Service) provides a framework used internally by Oracle components that makes it straightforward to add this instrumentation, and enables the results to be viewed in a variety of places. More information about DMS can be found in the Oracle Fusion Middleware Performance and Tuning Guide.

Things to Monitor During Testing

During testing it is expected to set the JVM heap size high to accommodate as much user load as possible. However, space allocated to the JVM takes away space from other processes, possibly causing them to excessively page fault or be swapped out.

Another important thing to monitor is heap size and garbage collection in the JVM. You'll want to confirm that the heap size is appropriate to handle the load, and whether there's a serious memory leak that is affecting performance over time.

You'll also want to monitor your back end systems, such as your database server.

Page Faulting and Swapping

The operating system is unlikely to be running optimally if free memory drops below 25MB per core, and in general free memory should average around 50MB per core to give the operating system "breathing space", given that its needs fluctuate over time. Sometimes free memory drops quite low as cached memory grows high. However, cached memory should expire relatively soon and be moved into free memory, so you'll need to monitor the combination of free and cached memory to spot if the operating system is too low on "breathing space."

When the operating system gets more desperate to obtain free memory it will swap out processes entirely. This is a reasonable thing to do if the processes aren't needed during the load test. If there are processes being swapped out during the entire test, you should investigate whether these processes can be permanently removed. If the swapped out processes are used during the load test and are swapping in/out routinely, this will have a very negative effect on performance. In that case you'll want to rebalance by lowering the heap size of the JVM, or by moving some processes to another machine until the swapping activity stops.

On Unix and Linux, you can use 'top' and 'vmstat' continuously to observe free space and swapping while the test is running. On Windows, you can use the built-in Task Manager; even more powerful are the Process Explorer and VMMap tools that are part of the Sysinternals Suite, currently offered as a free download from technet.microsoft.com.

```
top - 14:51:57 up 101 days, 18:31,  4 users,  load average: 1.53, 1.51, 1.17
Tasks: 126 total,   1 running, 125 sleeping,   0 stopped,   0 zombie
Cpu(s): 17.0% us,  7.0% sy,  0.0% ni, 46.9% id, 28.9% wa,  0.2% hi,  0.0% si
Mem:   4096132k total,  4063232k used,    32900k free,   559028k buffers
Swap:  4144464k total,   143580k used,  4000884k free,   760500k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31645 shalin    16   0 1678m 348m  26m S 31.7  8.7   0:13.73 java
31447 shalin    16   0 1742m 868m  37m S  3.0 21.7   1:00.30 java
31183 shalin    16   0 1686m 571m  25m S  1.7 14.3   0:25.63 java
31031 shalin    18   0 1695m 558m  28m S  1.3 14.0   0:25.86 java
29576 shalin    15   0 20484  13m 1844 S  0.7  0.3  60:20.77 Xvnc
   39 root      15   0     0    0    0 S  0.3  0.0  20:28.62 kswapd0
29747 shalin    15   0  113m  87m 7516 S  0.3  2.2  30:33.63 gnome-terminal
    1 root      16   0  3148  472  416 S  0.0  0.0   0:05.05 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:34.72 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:01.28 ksoftirqd/0
    4 root       5 -10     0    0    0 S  0.0  0.0   0:01.10 events/0
    5 root       5 -10     0    0    0 S  0.0  0.0   0:00.00 khelper
    6 root       5 -10     0    0    0 S  0.0  0.0   0:00.00 kthread
    7 root       5 -10     0    0    0 S  0.0  0.0   0:00.00 xenwatch
    8 root       5 -10     0    0    0 S  0.0  0.0   0:00.00 xenbus
   15 root      RT -10     0    0    0 S  0.0  0.0   0:32.53 migration/1
   16 root      34  19     0    0    0 S  0.0  0.0   0:01.48 ksoftirqd/1
```

Figure 3 – Sample Output from 'top' Command

```
#> vmstat 2 6
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 0  0   2536  21496 185684 1353000    0    0     0    14    1     2  0  0 100  0
 0  0   2536  21496 185684 1353000    0    0     0    28 1030   145  0  0 100  0
 0  0   2536  21496 185684 1353000    0    0     0     0 1026   132  0  0 100  0
 0  0   2536  21520 185684 1353000    0    0     0     0 1033   186  1  0  99  0
 0  0   2536  21520 185684 1353000    0    0     0     0 1024   141  0  0 100  0
 0  0   2536  21584 185684 1353000    0    0     0     0 1025   131  0  0 100  0
#>
```

Figure 4 – Sample Output from 'vmstat' Command

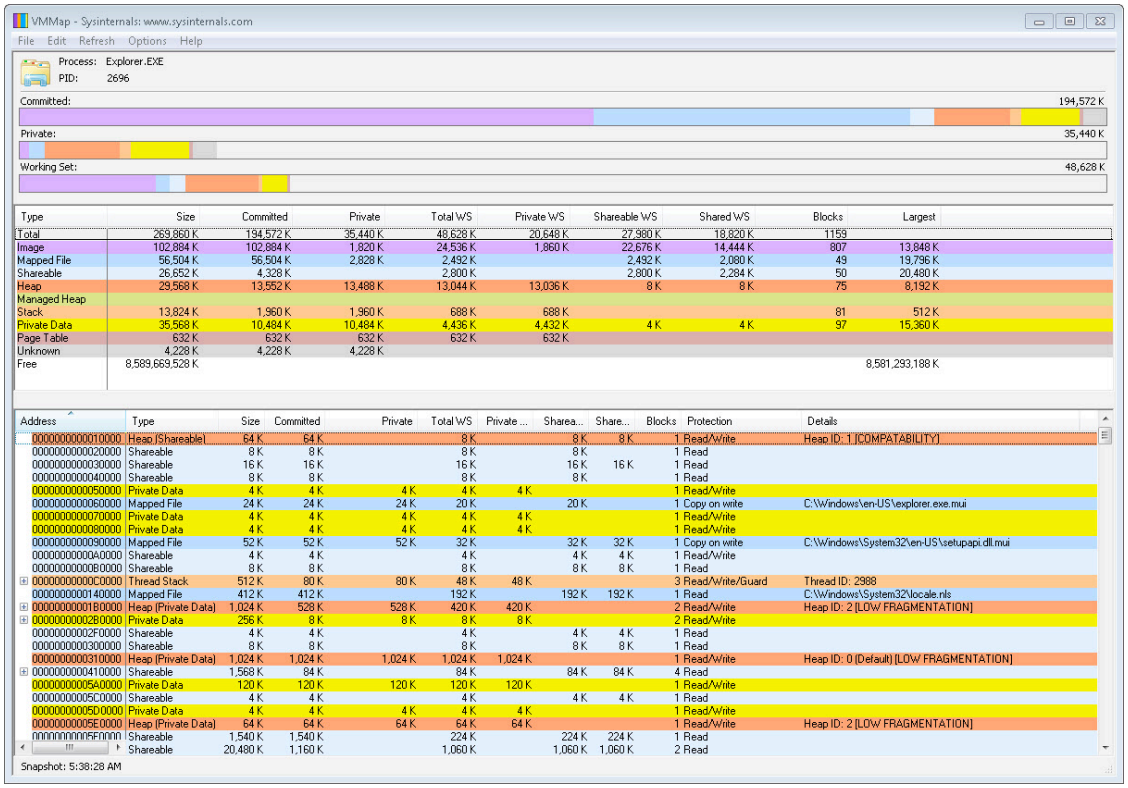

Figure 5 – Screenshot from the Sysinternals VMMap Tool

Heap Consumption

You'll want to set the heap size high on the JVM to handle a large number of users. But it's difficult to predict how much you will need, and as the application evolves its memory consumption needs will change, too. If you allocate too much, you can starve the other processes of memory, as discussed above. If you allocate too little, the JVM can spend too much time doing garbage collection, or can completely run out of memory. Heap usage is something you'll want to monitor during each run, and if it looks like too little heap is allocated, there's no point continuing. You'll want to abort the test, adjust the amount of heap allocated on the JVM command line, and restart.

As a general rule, "live memory" should remain below 70% of the maximum heap size. "Live memory" is the amount of heap shown in use immediately after a full garbage collection.

A common software defect is a memory leak, where memory gets allocated for objects that are never freed up. Given the complexity of modern software, minor leakage is all but inevitable. But if it is excessive, as it starts running out of memory the JVM will spend lots of time doing garbage collection, which will have a severe impact on performance. Eventually the JVM will exhaust all memory and will abort.

A simple tool for monitoring heap consumption is 'JConsole', which comes with the JVM. It has a handy button to interactively force a full garbage collection so you can see how much live memory is in use:

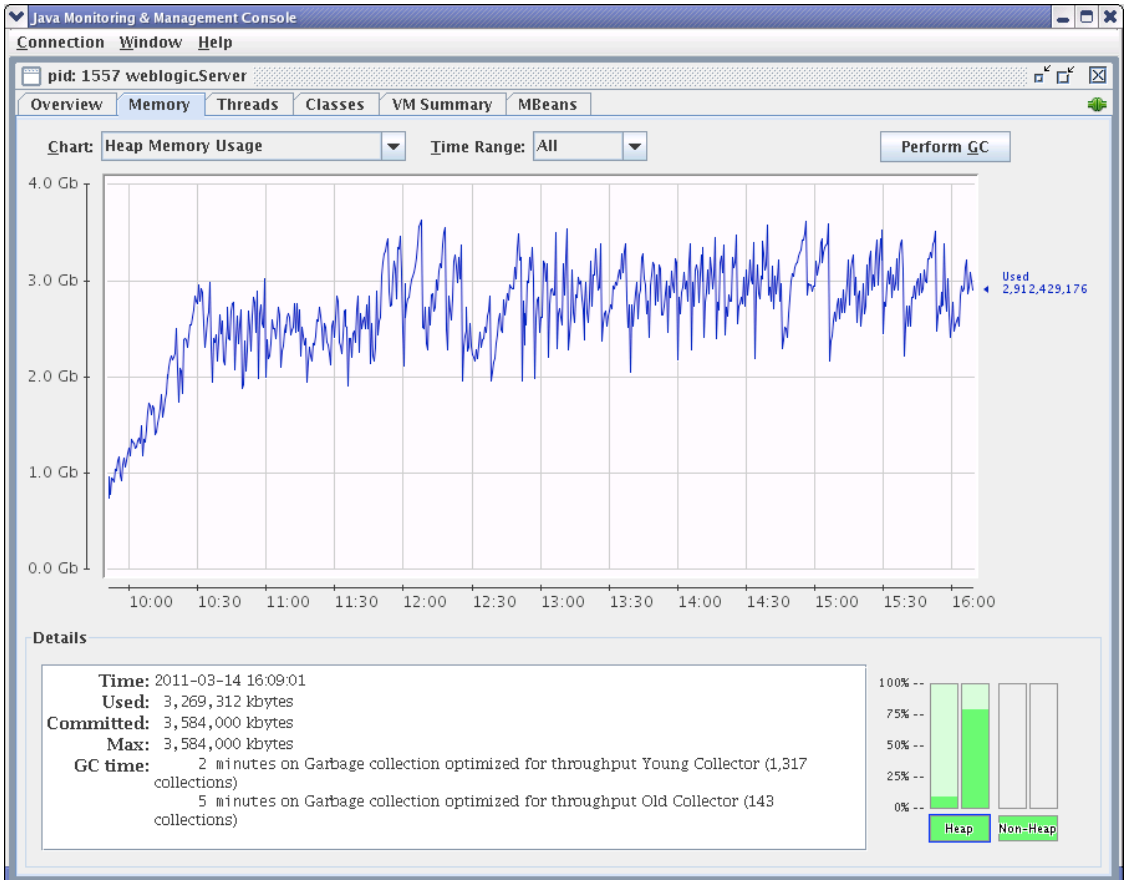

Figure 6 – Screenshot of JConsole

A more powerful tool for monitoring and debugging memory leaks is JRockit Mission Control:

Figure 7 – Screenshot of JRockit Mission Control

As the application starts up, memory consumption grows upward, but should level off after the first hour or so when the application has completely warmed up. As shown in Figures 6 and 7, memory consumption is never truly level – as the application runs, memory gets allocated and freed, but the memory usage grows until the garbage collector kicks in and coalesces freed memory. It's this periodic garbage collection that causes the saw tooth pattern to the peak memory allocation graph.

You can visually spot if more heap than absolutely necessary is allocated because the peak allocation may never hit the set maximum, and the differences in height between the peaks and valleys tend to be quite large. If too little heap is allocated, the peaks are routinely pegged at the maximum, the difference in height between the peaks and valleys are not far apart, and garbage collection is very frequent as the JVM tries to find some breathing room.

If there is a memory leak, the bottoms of the valleys will grow higher over time:

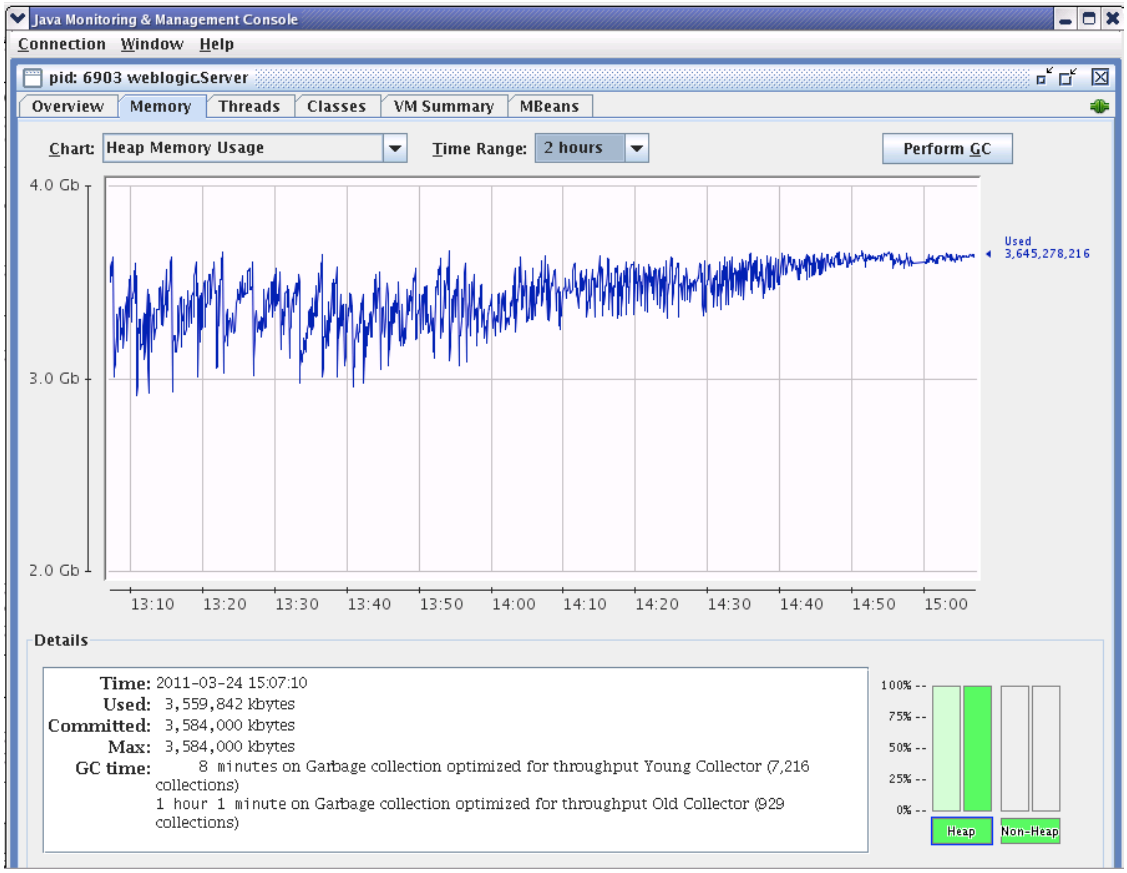

Figure 8 – A Screenshot of JConsole Showing a Memory Leak

DMS Sensors

You can monitor yours and Oracle's built-in DMS sensors in the application through various techniques. Popular methods include the DMS Spy Servlet or Oracle Enterprise Manager.

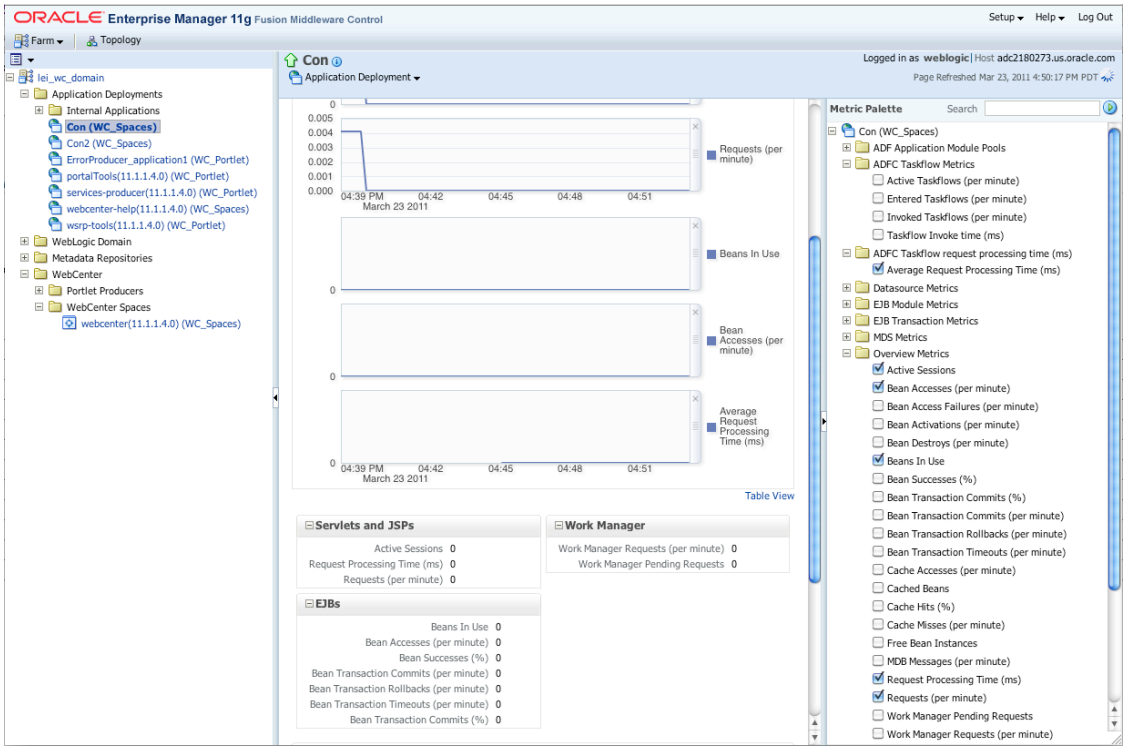

Figure 9 – Enterprise Manager Display for DMS Measurements

Log Files

With the application under load, there's a much greater chance of encountering concurrency and race conditions that weren't found before. The log files should be monitored during testing for exceptions that indicate serious failures in the application. Serious failures can cause your click streams to not work right, and/or slow performance significantly.

Deadlocks

It's relatively common to end up with a blocked thread because it can't obtain the lock to some resource. This is almost always an application design problem. It's important to monitor for this situation, as it will interfere with your testing. Your load execution software should give you an indication of the problem with responses that never complete. To debug further, you can use the Threading mbean in JConsole or Oracle Enterprise Manager, or take a thread dump and analyze the results, perhaps with the help of a thread dump analysis tool.

Back End Systems

Performance bottlenecks often lie in the back end systems, so it's important to monitor them during the load testing using tools and techniques appropriate for the particular system. The Oracle Database

Performance Tuning Guide is a good starting point for learning more about database monitoring and tuning.

Memory Consumption Analysis

There are often many causes of performance problems, from poor choices in configuration settings to non-optimal SQL queries, and the serious issues must be located and corrected. One area that often surprises people, but which always seems to keep applications from scaling up as expected, is excessive memory consumption. The main possibilities:

- System failure due to exhausted heap, either due to memory leak or excessive amounts of state being maintained
- Performance loss due to allocation/de-allocation of large amounts of memory or large numbers of objects

There aren't a lot of guidelines to help figuring out why the memory consumption is high; it mostly comes down to good detective work. There are some powerful tools at your disposal, including:

- JDeveloper Memory Profiler
- JRockit Mission Control
- Eclipse Memory Analyzer Tool

Undoubtedly you will find a personal favorite, but it makes sense to try them all, as each has unique features that may prove useful to your particular situation.

Response Times

The load generation tool you use will provide overall average and maximum response times, and will also pinpoint which of the specific request/response interactions in your click streams are the slowest. You should probe slow request/response interactions using extra instrumentation, or by debugging them through manual testing to understand where the time is being spent.

Conclusion

If you conduct performance/scalability testing and stress testing early and often, using appropriate tools and techniques, you will be able to find critical configuration and application design problems before you go live, giving you much higher confidence of a successful initial launch. By continuing to use these tools throughout the application lifecycle, especially before deploying maintenance releases, you should be able to spot serious performance and memory leak regressions before subjecting real users to these problems.

Appendix A – How to Preserving ADF State Using Oracle Application Testing Suite

It is very important that ADF view and controller states be preserved as the click stream is executing. Oracle ATS Complete Install comes with the Functional Testing Accelerator for Application Development Framework that will handle this automatically.

When creating a new project in OpenScript, choose the "Oracle Fusion/ADF" option in the Load Test Group. This will automatically apply the correlation library for ADF:

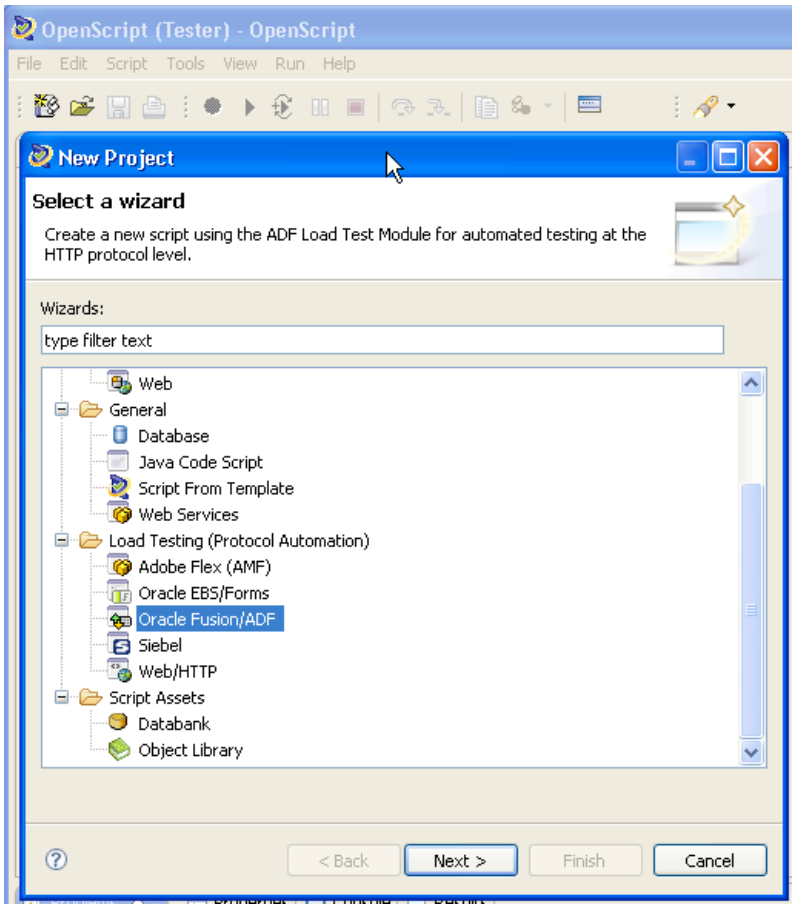

Figure 10 – OpenScript New Project Wizard

Also note there are APIs exclusively for ADF applications in the "adfload" and "http" classes.

Appendix B – How to Preserving ADF State Using HP LoadRunner

It is very important that ADF view and controller states be preserved as the click stream is executing. With HP Loadrunner, this is achieved through "correlation". The following is a sample correlation file that preserves view and controller state for ADF:

```
<?xml version="1.0"?>
<CorrelationSettings><Group Name="WebCenter" Enable="1" Icon="logo_default.bmp"><Rule
Name="adf.ctrl-state" LeftBoundText="_adf.ctrl-state=" LeftBoundType="1"
LeftBoundInstance="0" RightBoundText="&quot;" RightBoundType="1"
AltRightBoundText="Newline Character" AltRightBoundType="4" Flags="137"
ParamPrefix="adf.ctrl-state" Type="8" SaveOffset="0" SaveLen="-1" CallbackName=""
CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule Name="adf.winId"
LeftBoundText="_adf.winId=" LeftBoundType="1" LeftBoundInstance="0"
RightBoundText="&amp;" RightBoundType="1" AltRightBoundText="" AltRightBoundType="1"
Flags="137" ParamPrefix="adf.winId" Type="8" SaveOffset="0" SaveLen="-1"
CallbackName="" CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
Name="jsessionid" LeftBoundText="jsessionid=" LeftBoundType="1" LeftBoundInstance="0"
RightBoundText="&quot;" RightBoundType="1" AltRightBoundText="" AltRightBoundType="1"
Flags="136" ParamPrefix="jsessionid" Type="8" SaveOffset="0" SaveLen="-1"
CallbackName="" CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
Name="STATETOKEN" LeftBoundText="javax.faces.ViewState&quot; value=&quot;"
LeftBoundType="1" LeftBoundInstance="0" RightBoundText="&quot;&gt;" RightBoundType="1"
AltRightBoundText="" AltRightBoundType="1" Flags="136" ParamPrefix="STATETOKEN"
Type="8" SaveOffset="0" SaveLen="-1" CallbackName="" CallbackDLLName="" FormField=""
ReplaceLB="" ReplaceRB=""/><Rule Name="afrLoop" LeftBoundText="_afrLoop="
LeftBoundType="1" LeftBoundInstance="0" RightBoundText="&quot;" RightBoundType="1"
AltRightBoundText="" AltRightBoundType="1" Flags="136" ParamPrefix="afrLoop" Type="8"
SaveOffset="0" SaveLen="-1" CallbackName="" CallbackDLLName="" FormField=""
ReplaceLB="" ReplaceRB=""/><Rule Name="adf.ctrl-state_new" LeftBoundText="_adf.ctrl-
state=" LeftBoundType="1" LeftBoundInstance="0" RightBoundText="&lt;"
RightBoundType="1" AltRightBoundText="" AltRightBoundType="1" Flags="9"
ParamPrefix="adf.ctrl-state_new" Type="8" SaveOffset="0" SaveLen="-1" CallbackName=""
CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
Name="SecurityUsersCreateUserPortletfrsc"
LeftBoundText="SecurityUsersCreateUserPortletfrsc&quot; value=&quot;" LeftBoundType="1"
LeftBoundInstance="0" RightBoundText="&quot;&gt;" RightBoundType="1"
AltRightBoundText="" AltRightBoundType="1" Flags="8"
ParamPrefix="SecurityUsersCreateUserPortletfrsc" Type="8" SaveOffset="0" SaveLen="-1"
CallbackName="" CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
Name="CreatedPageName" LeftBoundText="/Page" LeftBoundType="1" LeftBoundInstance="0"
RightBoundText=".jspx" RightBoundType="1" AltRightBoundText="" AltRightBoundType="1"
Flags="1037" ParamPrefix="createPageName" Type="8" SaveOffset="0" SaveLen="-1"
CallbackName="" CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
Name="adfp_rendition_cahce_key" LeftBoundText="_adfp_rendition_cahce_key="
LeftBoundType="1" LeftBoundInstance="0" RightBoundText="&amp;" RightBoundType="1"
AltRightBoundText="" AltRightBoundType="1" Flags="8"
ParamPrefix="adfp_rendition_cahce_key" Type="8" SaveOffset="0" SaveLen="-1"
CallbackName="" CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
Name="adfp_request_hash" LeftBoundText="_adfp_request_hash=" LeftBoundType="1"
LeftBoundInstance="0" RightBoundText="&quot;" RightBoundType="1" AltRightBoundText=""
AltRightBoundType="1" Flags="8" ParamPrefix="adfp_request_hash" Type="8" SaveOffset="0"
SaveLen="-1" CallbackName="" CallbackDLLName="" FormField="" ReplaceLB=""
ReplaceRB=""/><Rule Name="adfp_full_page_mode_request"
LeftBoundText="_adfp_full_page_mode_request%3D" LeftBoundType="1" LeftBoundInstance="0"
RightBoundText="%" RightBoundType="1" AltRightBoundText="" AltRightBoundType="1"
Flags="8" ParamPrefix="adfp_full_page_mode_request" Type="8" SaveOffset="0" SaveLen="-
1" CallbackName="" CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
Name="adfp_full_page_mode_request2" LeftBoundText="_adfp_full_page_mode_request="
LeftBoundType="1" LeftBoundInstance="0" RightBoundText="&amp;" RightBoundType="1"
AltRightBoundText="" AltRightBoundType="1" Flags="8"
ParamPrefix="adfp_full_page_mode_request2" Type="8" SaveOffset="0" SaveLen="-1"
CallbackName="" CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule
```

Name="afrLoop2" LeftBoundText="_afrLoop=" LeftBoundType="1" LeftBoundInstance="0"
RightBoundText="<" RightBoundType="1" AltRightBoundText="" AltRightBoundType="1"
Flags="8" ParamPrefix="afrLoop2" Type="8" SaveOffset="0" SaveLen="-1" CallbackName=""
CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule Name="afrLoop3"
LeftBoundText="_afrLoop=" LeftBoundType="1" LeftBoundInstance="0"
RightBoundText="&" RightBoundType="1" AltRightBoundText="" AltRightBoundType="1"
Flags="136" ParamPrefix="afrLoop3" Type="8" SaveOffset="0" SaveLen="-1" CallbackName=""
CallbackDLLName="" FormField="" ReplaceLB="" ReplaceRB=""/><Rule Name="wsrp-
resourceState" LeftBoundText="wsrp-resourceState~25253D" LeftBoundType="1"
LeftBoundInstance="0" RightBoundText="~252526" RightBoundType="1" AltRightBoundText=""
AltRightBoundType="1" Flags="8" ParamPrefix="wsrp-resourceState" Type="8"
SaveOffset="0" SaveLen="-1" CallbackName="" CallbackDLLName="" FormField=""
ReplaceLB="" ReplaceRB=""/><Rule Name="_afrWindowMode_checksum"
LeftBoundText="_afrWindowMode~253D0~26checksum~3D" LeftBoundType="1"
LeftBoundInstance="0" RightBoundText="/container-view" RightBoundType="1"
AltRightBoundText="" AltRightBoundType="1" Flags="8"
ParamPrefix="_afrWindowMode_checksum" Type="8" SaveOffset="0" SaveLen="-1"
CallbackName="" CallbackDLLName="" FormField="" ReplaceLB=""
ReplaceRB=""/></Group></CorrelationSettings>

White Paper Title
March 2011
Author: Stewart Wilson
Contributing Authors: Shaun Lin

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together