Performance testing and analysis with WebSphere Application Server

David Hare

August 01, 2012

IBM® WebSphere® Application Server supports an ever-growing range of applications, each with their own unique set of features, requirements, and services. Proper performance testing and analysis on each of these applications is essential to ensure they are performing at their maximum potential. This article provides guidance on some best practices on how to build a performance test, compare results across application or environment changes, and how to identify bottlenecks using freely available tools from IBM. The methodologies described here apply to all versions of WebSphere Application Server, including the newly released WebSphere Application Server V8.5.

Introduction

Can you relate to any of these statements:

- *Currently we're not doing any performance testing. We'd like to but we're just not sure where to even begin.*
- *Our application was running fine, but after the development team sent us an updated version to deploy, we're seeing much higher CPU usage on the server. Where is the high CPU usage is coming from?*
- *We migrated our application from version 2.0 to 3.0 and now response times are three times longer. What is causing these delays?*
- *Our application needs to support 40% more users in the next three months. How can we prepare for that other than simply adding more machines to the cluster?*

These statements represent very common scenarios, so if any of these sound familiar, you're not alone. The goal of this article is to address these types of situations by suggesting some best practices with regard to basic procedures for conducting tests and useful tools that are available. At a high level, the major topics discussed here will help you:

- Write useful performance test cases.
- Drive varying amounts of load to stress low and high utilization.
- Record key performance and system metrics.
- Conduct performance testing in parallel with application development cycles.
- Use tools like IBM Health Center to find performance bottlenecks.

- Work with development teams to fix bottlenecks and re-measure.

The big deal about performance testing

At some point along the way, performance testing got penciled in as something that is done just before rolling into production. It's frequently a very minimal effort with not enough time allotted to identify and fix real problems that eventually will show up in the production environment. The universally recommended approach to proper performance testing is to implement the performance lifecycle, in which performance testing is scheduled as part of the development work, iteratively testing as new features are integrated. This enables bottlenecks to be identified and resolved with plenty of time left in the release cycle before everything is rolled into production.

Another benefit of proper performance testing is the opportunity to tune the environment (operating system, JVM, application server, application, database, and so on) for maximum performance. Only through proper performance testing can tuning parameters be assessed to determine if they are providing any value. Many users set JVM heap sizes based on developer recommendations and don't tune anything else because it isn't considered necessary. You might be surprised to learn that it might be possible to cut the amount of hardware needed for an un-tuned environment in half just by conducting some simple tuning steps. This article proves the point.

With some simple tuning procedures, the DayTrader performance benchmark application can handle over 2x the load as the un-tuned environment. This means the same number of users could potentially be supported with half the available hardware resources. Think about the costs that could save.

In addition to iterative testing throughout the development cycle and the benefit of testing for tuning purposes, the other major advantage of extensive performance testing is the ability to compare results across application and environment changes. Real performance testing records key metrics (discussed later) that enable administrators to gain insight into problems that might be arising. To go back to one of the common comments above, many users aren't prepared to figure out where a problem is coming from when they migrate to a newer version of an application because they never did proper performance testing or recorded key system metrics for the earlier version. Without it, a test server with the earlier application version will likely need to be setup as a comparison point. Having this type of data makes analysis of where the regression is coming from much easier to find.

Best practices for proper performance testing

There are two fundamental best practices for proper performance testing that can be summed as follows:

- **Vary the number of users (or client load).** A production environment typically has a varying number of active users throughout the day. Quality testing ensures the application performs well under small loads and peak (for example, Black Friday) loads. This might mean changing around the “think” time in between requests and changing around the active number of users hitting the application. One of the best ways to do this is to perform a test with 1 active user, 2 users, 4 users, 8 users, and so on. You will see this in practice later.

- **Record key system and performance metrics.** There are several important metrics that should be recorded for all scenarios. For each performance run, the most important metrics to record are:
  - Throughput (requests per second)
  - Response times
  - Application server machine CPU utilization %
  - Other machine CPU utilization % (web server, load driver, database, as applicable)

The throughput and response time metrics can be seen in whichever load driving tool is being used. The CPU utilization, memory utilization, disk I/O, and network traffic metrics can be seen with tools like vmstat or nmon on Linux® or AIX®, or Task Manager on Windows®. In addition to the above metrics, it is also important to record all system level information. This includes operating system level, number of active cores, how much memory (RAM) is available, the “Java™ version” output, the WebSphere Application Server version information, and all tuning that has been applied. Recording all of these metrics will enable you to quickly compare scenarios, even if the scenarios being compared were tested two years apart.

Many users don't have the hardware available to replicate their production environment with a testing environment of the same size. In these cases, the recommended approach is to scale the performance test based on the resources that are available. For example, assume a production environment consists of ten physical machines, each running two instances of WebSphere Application Server. If only one physical machine is available for performance work, this machine could be setup as identical as possible to the production machines, and the load driven in the performance test would be roughly 10% of the expected production workload. Eventually the performance test should be ramped up to replicate the full production environment. This way, other processes such as the database and LDAP are load tested as well.

Test cases and load drivers

The very first step in conducting performance tests and finding application bottlenecks is to write useful test cases. Results and analysis are only as good as the test case that was used to produce them, so this step should not be taken lightly. Stressing application code paths that users only hit less than 10% of the time is not nearly as beneficial as stressing code paths that all users will hit. Focus on the most popular code paths first, and build your tests down to the lesser utilized code paths later. Spend a lot of time here really investing in a quality test case that emulates actual user traffic. This developerWorks article is a great resource for getting started in writing performance tests cases.

After mapping out a test case concept, you'll need to put it into practice with a performance load driving tool. There are many load drivers available to choose from, including IBM Rational Performance Tester and Apache's open source Jmeter. We'll refer to the latter in this article.

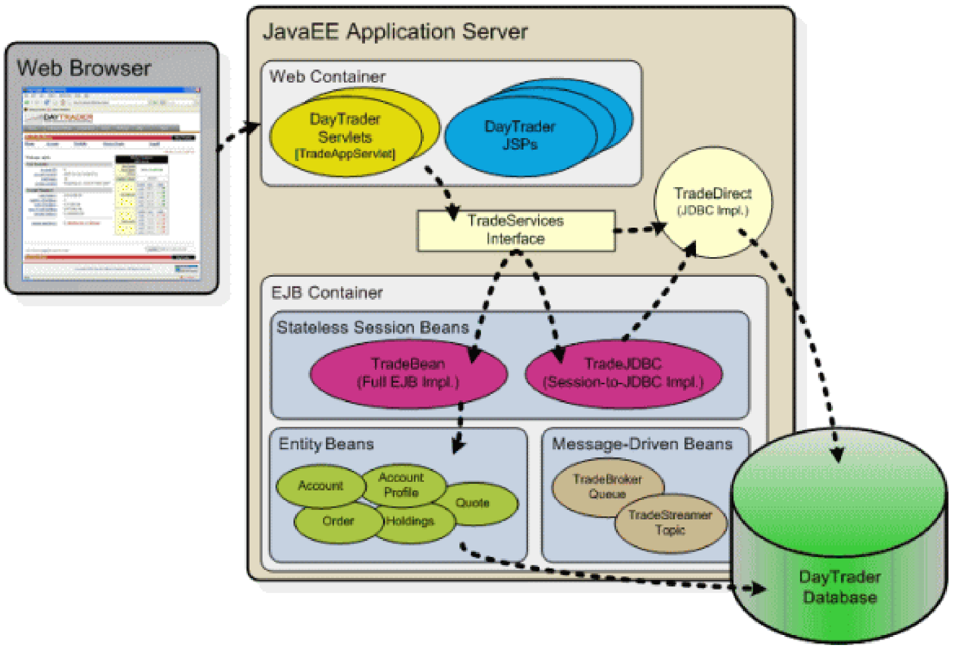
An example with DayTrader

Chances are good that if you've read any performance articles on developerWorks before, you've already heard of the “Trade” or “DayTrader” benchmark. The Apache DayTrader Performance

Benchmark Sample application simulates a simple stock trading application that lets users login/logout, view their portfolio, look up stock quotes, buy and sell stock shares, and manage account information. DayTrader not only serves as an excellent application for functional testing, but it also provides a standard set of workloads for characterizing and measuring application server and component level performance.

DayTrader is built on a core set of Java EE technologies that include Java servlets and JavaServer™ Pages (JSPs) for the presentation layer, and Java database connectivity (JDBC), Java Message Service (JMS), Enterprise JavaBeans (EJBs) and message-driven beans (MDBs) for the back end business logic and persistence layer. Figure 1 shows a high-level overview of the application architecture.

Figure 1. DayTrader application overview



IBM has published a sample DayTrader package for download which includes the DayTrader application and the required deployment descriptors that you can install on WebSphere Application Server V7.0 or newer releases.

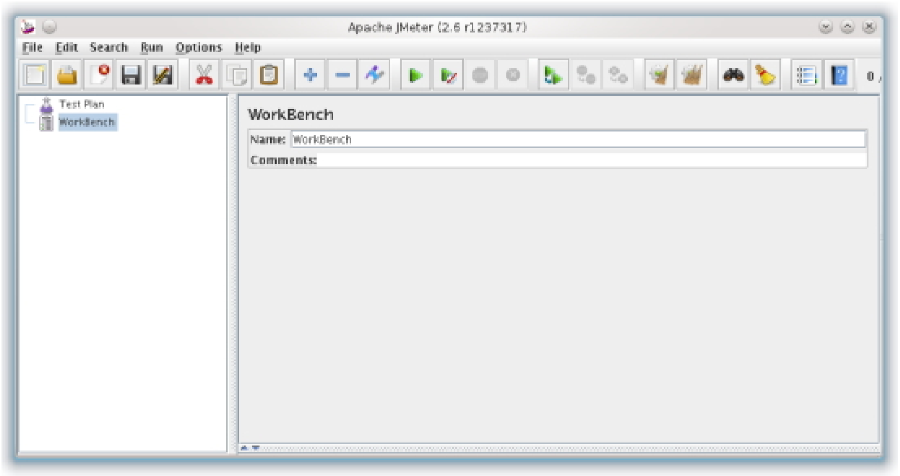
In this example, the DayTrader sample application was deployed to a base instance of WebSphere Application Server V8.5. One of the neat features of DayTrader is the **TradeScenarioServlet** link under the **Configuration** tab. This links to a servlet that emulates a population of web users by randomly generating a specific DayTrader operation for each user that accesses the servlet. (For example, one user might view their portfolio, one might perform a stock buy operation, one might look up a stock quote, and so on.) This ensures each of the main operations in DayTrader are executed during the test, and over time, because it's random, each operation should be executed roughly the same number of times. There are numerous resources available for how to use JMeter to write very complex performance tests where each of the operations could be specified exactly how many times to hit, and in what order, but for the purpose of this article the test case will be kept relatively simple and use this TradeScenarioServlet.

Using JMeter

To setup Jmeter and get it running to drive the performance test:

1. Install Jmeter. Pointing to your **java** directory, launch JMeter from the <JMeter_Home>/bin/ directory using the `jmeter.sh` or `jmeter.bat` script. You should see a panel similar to Figure 2.

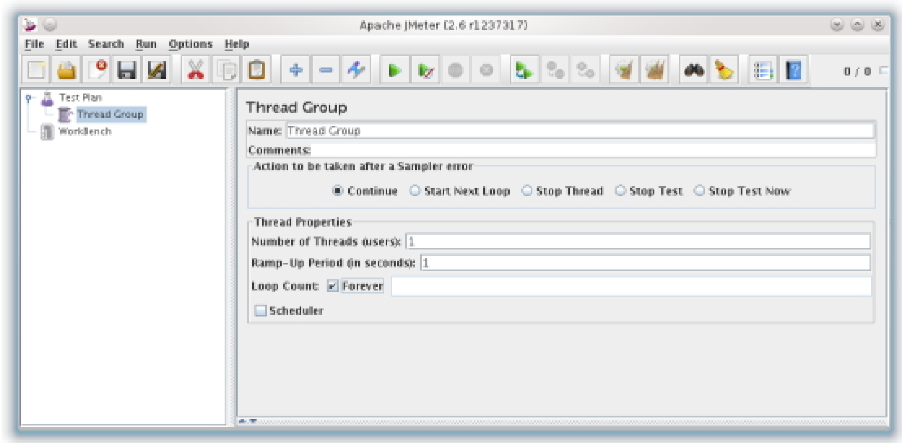
   Figure 2. JMeter default view

   
   Apache JMeter (2.6 r1237317)
File Edit Search Run Options Help
Test Plan
WorkBench

   WorkBench
   Name: WorkBench
   Comments:

2. Right click on **Test Plan** and go to **Add > Thread Group**. This is where you define the number of users to drive load with. For starters, use these values (Figure 3):
   - Number of Threads (users): `1`
   - Loop Count: **Forever**

   Figure 3. JMeter Thread Group view

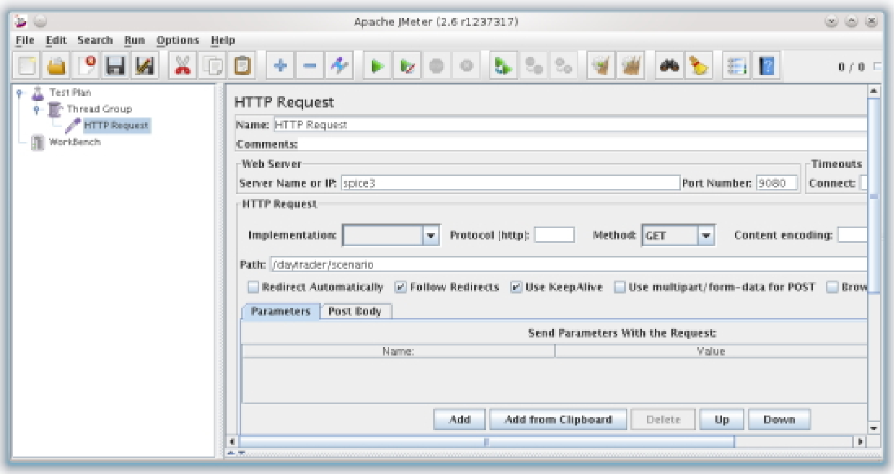
   
   Apache JMeter (2.6 r1237317)
File Edit Search Run Options Help
0 / 0
Test Plan
Thread Group
WorkBench

   Thread Group
   Name: Thread Group
   Comments:
   Action to be taken after a Sampler error
   [✓ Continue] Start Next Loop Stop Thread Stop Test Stop Test Now
   Thread Properties
   Number of Threads (users): 1
   Ramp-Up Period (in seconds): 1
   Loop Count: [✓ Forever]
   Scheduler

3. Right-click on the thread group you just created, and go to **Add > Sampler**. A sampler defines the type of load you want to drive. There are samplers for HTTP requests, JMS requests, Web services messages, and so on. The JMeter user manual documents each of the available samplers. Because DayTrader supports web-based traffic, this example uses

the **HTTP Request**. Fill out the values for the hostname, port, and path according to your environment (Figure 4).

Figure 4. HTTP request



Name: HTTP Request
Comments:
Web Server
Server Name or IP: spice3
Port Number: 9080
Timeouts
Connect:
HTTP Request
Implementation:
Protocol [http]:
Method: GET
Content encoding:
Path: /daytrader/scenario
☐ Redirect Automatically ✓ Follow Redirects ✓ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Brow

4. Right click on the HTTP request you just created, and select **Add > Timer**. A timer adds "think time" in between requests. This simulates a user clicking on a page, and then pausing to read some information on the page, before making another request. There are many predefined timers you can choose from that range in complexity from a constant fixed timer to a gaussian or poisson distributed timer. The JMeter user manual documents each of the available timers. For this simple example, just use a **Constant Timer** of 5 ms.
5. Right click on the thread group and go to **Add > Listener > Summary Report**. This will show you the response times and throughput results while the test is running.
6. Save the settings to a file so you can load them again later.

Running the test

Always make sure the application works through a browser before starting the load driving tool. When ready, click the green arrow or click **Run > Start**. JMeter should now be driving one client to the server path you specified, waiting 5 ms in between each request. If you click on the **Summary Report** you can view the results as the test runs.

You should notice the throughput increasing over time; this is called the "warm-up" period. The JRE needs some warm-up time to load all the classes and let the JIT make some optimizations. The throughput will ultimately stabilize and reach a fixed number (give or take a few requests per second). Depending on how complex your test is, this warm-up period could be 30 seconds or 30 minutes.

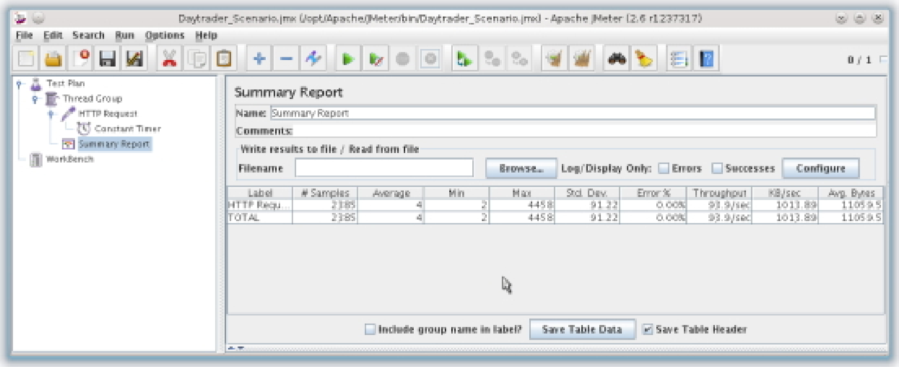
While the test is running, open a terminal (Linux or AIX) and run `vmstat 5` to display system metrics every five seconds (Listing 1).

Listing 1

```
[root@spice3 bin]# vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 8235164 104920 500956    0    0    13    11   44  154  8  5 86  0  0
 0  0      0 8235164 104928 500956    0    0     0     3 8030 4987  6  1 93  0  0
 1  0      0 8235040 104936 500956    0    0     0     3 7982 4944  5  1 94  0  0
 0  0      0 8233116 104936 500960    0    0     0     6 8126 5020  7  1 92  0  0
 0  0      0 8231068 104944 500960    0    0     0     6 7952 4939  6  1 93  0  0
 0  0      0 8231316 104952 500960    0    0     0     3 7761 4819  5  1 94  0  0
Example vmstat output showing ~7% CPU utilization.
```

If you're using Windows, right click on the task bar and select **Task Manager** and choose the **Performance** tab. Once the throughput in JMeter reaches a stable value, record the CPU utilization on the server running WebSphere Application Server and any other applicable servers, then stop the Jmeter test by clicking the red stop sign, or **Run > Stop**. The JMeter Summary Results view should look similar to Figure 5.

Figure 5. JMeter Summary Report view


Daytrader_Scenario.jmx (/opt/Apache/JMeter/bin/Daytrader_Scenario.jmx) - Apache JMeter (2.6 r1237317)

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
HTTP Requ...	2385	4	2	4458	91.22	0.00%	93.9/sec	1013.89	11059.5
TOTAL	2385	4	2	4458	91.22	0.00%	93.9/sec	1013.89	11059.5

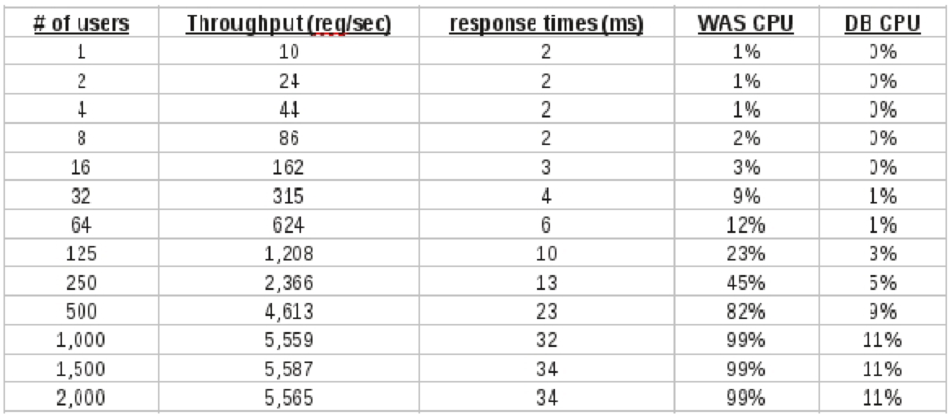
In a spreadsheet, record the throughput result (93.9 req/sec) and the average response time result (4 ms). The Min, Max, and Std. Dev. response time results can also be beneficial to record if you want even more detailed information.

After recording all the results, select **Run > Clear** to clear the Summary Report results. This concludes the test for the single user. Now simply repeat the above steps by increasing the number of users to 2, then 4, then 8, and so on. You should observe the throughput increases each time as you add more users. Eventually, you'll observe the throughput stops increasing and may even start to decrease. Once that plateau (and possible degradation) is reached, the test can be stopped.

Analyzing the results

After completing the above steps, you should have a spreadsheet that looks something like Figure 6. (These particular results are very much dependent on the DayTrader application and the environment in which this test was run. Your actual results will likely look very different.)

Figure 6. Test Results – Table view



# of users	Throughput (req/sec)	response times (ms)	WAS CPU	DB CPU
1	10	2	1%	0%
2	24	2	1%	0%
4	44	2	1%	0%
8	86	2	2%	0%
16	162	3	3%	0%
32	315	4	9%	1%
64	624	6	12%	1%
125	1,208	10	23%	3%
250	2,366	13	45%	5%
500	4,613	23	82%	9%
1,000	5,559	32	99%	11%
1,500	5,587	34	99%	11%
2,000	5,565	34	99%	11%

Having the raw data in a tabular format like this is very beneficial, but it's also helpful to view the results in a graphical format. One of the best ways to visualize this is to use an XY scatter plot. Building a graph to chart the results makes it much easier to identify trends. Figure 7 charts the throughput and WebSphere Application Server CPU % versus the number of clients.

Figure 7. Test Results – Graph

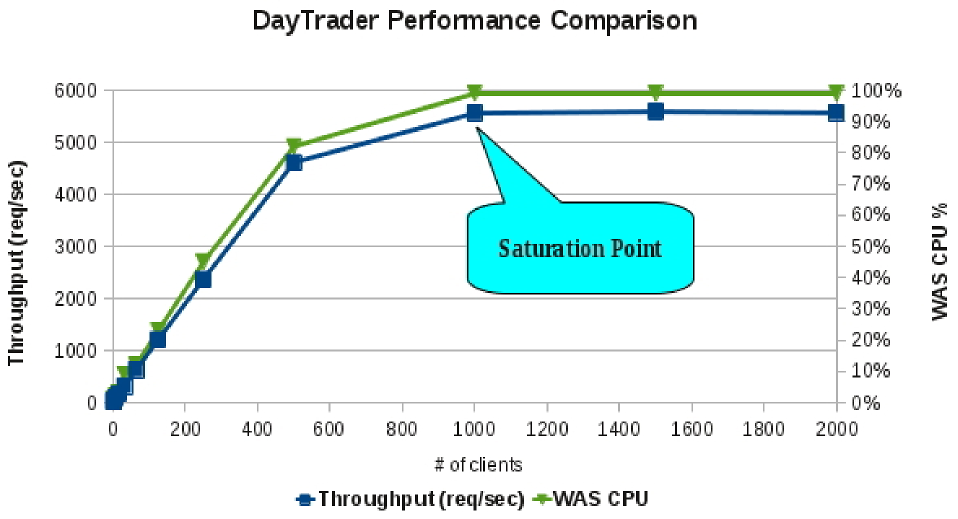


Figure 7 above shows some interesting characteristics. First, you see that the throughput curve and CPU % curve match closely together. Second, you see that the application throughput scales linearly from 1 client up to 500 clients. This is the desired result. However, somewhere in between 500 clients and 1,000 clients, the increase in throughput starts to slow down. (More tests could be done at this point with user loads in between 500 and 1,000 to find out exactly where this slowdown occurs.) Increasing the client load beyond 1,000 clients does not improve your overall application throughput. This is what's called the **saturation point**. This is a key value that must be found during the performance test. The saturation point tells you that you've reached your maximum capacity for this application, tuning, configuration, and environment. Adding more users beyond this point will only increase client response times, but will not increase the overall application throughput. To achieve better performance beyond this point, an application code change, tuning change, or environmental change must occur.

DayTrader versus your application

DayTrader is not representative of all applications. Your application might reach the saturation point much sooner. If that is the case, it's likely indicative of an application bottleneck, which requires application analysis to remediate.

This type of testing and analysis is paramount in sizing and capacity planning discussions. It is only by identifying the saturation point that you can accurately estimate the total capacity needed to support a production environment. Too often, someone will say, “I need to support 10,000 users in this environment” and then run tests with that client load. Typically, this approach leads to a variety of overloaded conditions in one or more components, either in WebSphere Application Server or in other infrastructure components (network, database, and so on). A more productive approach is to determine what is achievable in terms of client load and throughput with a single application server and then proceed with run time and application tuning based on this. Once tuning is complete, you can then turn your attention to determining how many application server processes and physical servers are required to satisfy the scalability requirements.

Save a new test in JMeter with the # of Threads (users) you found as your saturation point. This can be used as a quick test to compare performance as you make changes to your application or environment, without going through the entire scalability test again. This isn't to say you should no longer execute the scalability test, but running with just the load at the saturation point is a great place to compare changes that likely won't show any difference at lower loads where the CPU is not fully utilized. A good practice is to run the saturation point load for minor application or tuning changes, and to repeat the entire scalability test for major application or environment changes.

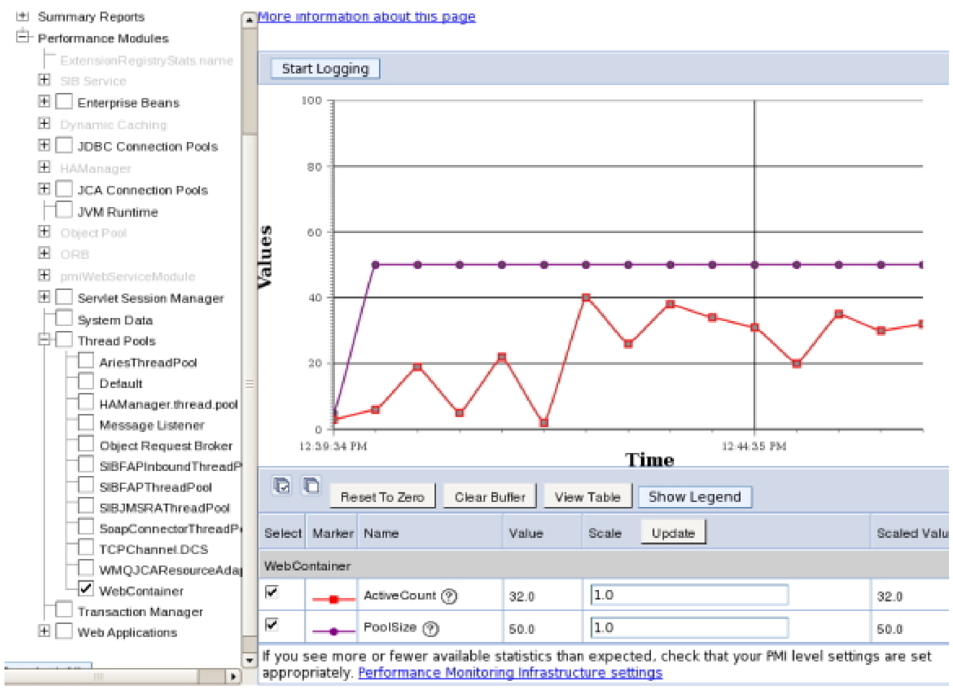
Performance tools

The sections above are prerequisites to doing any real analysis work. You must first understand how to generate repeatable performance results before looking for bottlenecks and performance improvements. Now that you're ready to start looking for improvements, here are two performance tools with which you should start your analysis:

- IBM Tivoli® Performance Viewer
  Tivoli Performance Viewer (in the admin console) is a very useful tool for monitoring WebSphere Application Server. This article really highlights the benefits of using Tivoli Performance Viewer to optimally tune an environment. In the same manner, Tivoli Performance Viewer can be used to quickly check if there are any bottlenecks that could easily be removed by tuning.
  Here are some simple steps to get started with Tivoli Performance Viewer:
  1. Re-start the JMeter load with the number of users that were identified as the saturation point.
  2. Login to the administrative console and select **Monitoring and Tuning > Performance Viewer > Current Activity**. Click on the server you want to monitor, and then expand **Performance Modules**. This will display a list of Performance Modules that are available to view.
  3. Your application characteristics will determine which of these modules make the most sense to view. For the DayTrader example, or any other web-based traffic, start with the

Thread Pools > WebContainer module. After checking that box, click the **View Module** button at the top. This will display a graph that looks something like Figure 8, populating more data automatically every 30 seconds as the JMeter test continues to run.

Figure 8. WebContainer PMI data



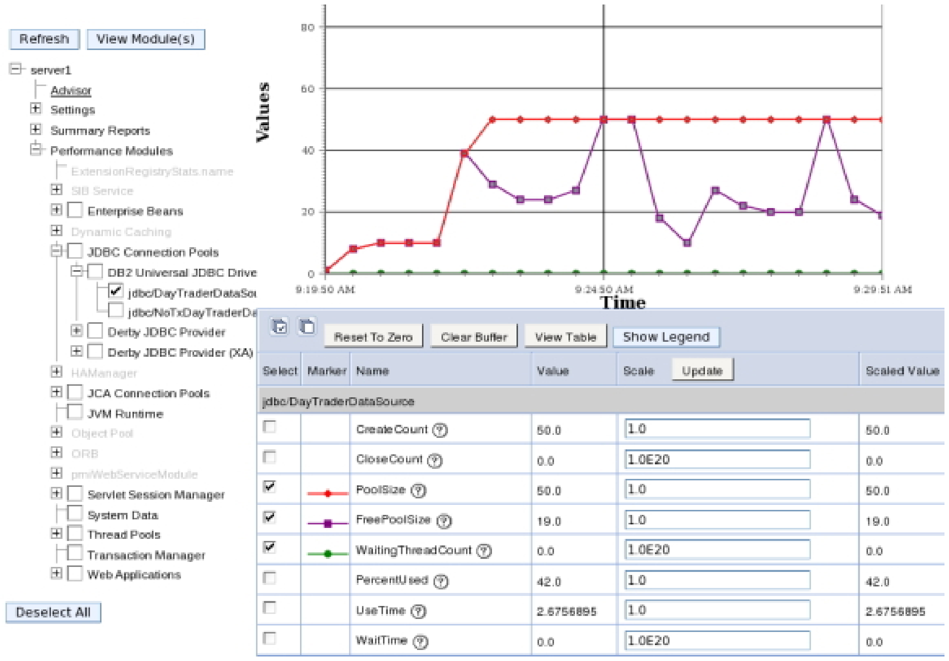
Select	Marker	Name	Value	Scale	Update	Scaled Valu
WebContainer						
✓		ActiveCount	32.0	1.0		32.0
✓		PoolSize	50.0	1.0		50.0

If you see more or fewer available statistics than expected, check that your PMI level settings are set appropriately. Performance Monitoring Infrastructure settings

This example shows that the WebContainer thread pool size is at 50 threads, while approximately 32 are in use. This tells you that the WebContainer thread pool size is not a bottleneck at the current workload. If the active count was fluctuating between 45-50, then the WebContainer thread pool size could be a bottleneck. In that case, it would be best to increase the WebContainer thread pool size and repeat the test to see if the performance improves. If the throughput does increase, you probably want to re-run the full scalabiltiy test again to re-establish your baseline.

4. Continue repeating step 3 with other modules that are applicable to your application. For a transactional application like DayTrader, another module you should view is the JDBC Connection Pool information. Expand **JDBC Connection Pools > *(your JDBC driver)*** and select your datasource JNDI name. Click the **View Module** button to display a graph that looks something like Figure 9.

Figure 9. DataSource PMI data



Select	Marker	Name	Value	Scale	Scaled Value
jdbc/DayTraderDataSource					
☐		CreateCount	50.0	1.0	50.0
☐		CloseCount	0.0	1.0E20	0.0
✓	🟥	PoolSize	50.0	1.0	50.0
✓	🟪	FreePoolSize	19.0	1.0	19.0
✓	🟩	WaitingThreadCount	0.0	1.0E20	0.0
☐		PercentUsed	42.0	1.0	42.0
☐		UseTime	2.6756895	1.0	2.6756895
☐		WaitTime	0.0	1.0E20	0.0

This chart shows a few different options charted rather than the default selections. The PoolSize, FreePoolSize, and WaitingThreadCount are all great metrics to review to ensure your connection pool isn't a source of contention and WebContainer threads aren't queued up waiting for a connection to the database. In the example above, the connection pool size is fixed at 50 connections (this is a tuning setting). The free pool size is fluctuating around 20, meaning that roughly 30 connections are active at a time. Together, this produces a Waiting Thread Count of 0, meaning that no WebContainer threads are waiting for a connection to the database. This verifies that the JDBC connection pool size is not a bottleneck. If the free pool size is 0 and the waiting thread count is greater than 0, then you might want to repeat the test with a higher connection pool size.

Continue this process with any other performance modules that are beneficial for monitoring your application. The WebSphere Contrarian: Preparing for failure has an extensive list of statistics that can be of great value to monitor. Once this exercise has been completed, you can move onto more detailed analysis with the IBM Health Center.

- IBM Monitoring and Diagnostic Tools for Java - Health Center
  The IBM Monitoring and Diagnostic Tools for Java - Health Center (hereafter referred to as Health Center, which is part of the IBM Support Assistant) tool is the recommend tool for detailed performance analysis on a WebSphere Application Server process. Health Center provides a wealth of knowledge about the performance of a server, including information about:
  - Memory usage
  - Garbage collection statistics
  - Method level profiling

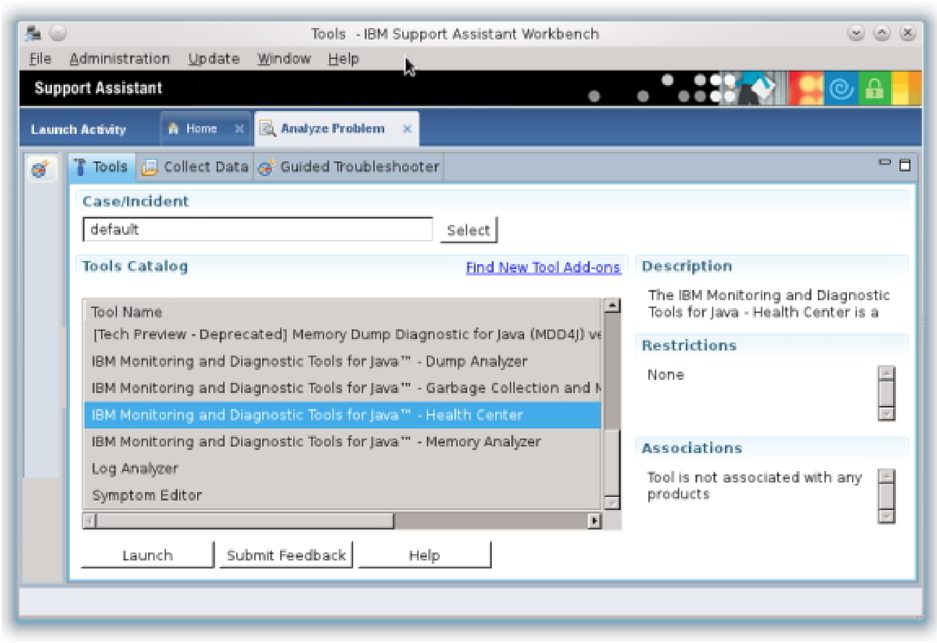
- Lock contention analysis.

Health Center is included as a tool in IBM Support Assistant (ISA), which is freely downloadable. Make sure you install ISA to a different machine than the application server machine, otherwise Health Center will take up resources away from the application server process, and your results might not be accurate. Health Center can be ran in an interactive mode, and in a “headless” mode where the information is saved to a file for later viewing. For this example, you'll use the interactive mode.

To launch Health Center:

1. Restart the WebSphere Application Server process you want to get detailed information on with the generic JVM argument `-Xhealthcenter`.
2. Start ISA. When loaded, select **Analyze Problem**. (If you have not previously done so, you will need to tell ISA you are interested in tools for WebSphere Application Server.)
3. Select **IBM Monitoring and Diagnostic Tools for Java – Health Center** and click **Launch** (Figure 10)

Figure 10. Launching Health Center from ISA



Case/Incident

default

Tools Catalog

Tool Name
[Tech Preview - Deprecated] Memory Dump Diagnostic for Java (MDD4J) ve
IBM Monitoring and Diagnostic Tools for Java™ - Dump Analyzer
IBM Monitoring and Diagnostic Tools for Java™ - Garbage Collection and M
IBM Monitoring and Diagnostic Tools for Java™ - Health Center
IBM Monitoring and Diagnostic Tools for Java™ - Memory Analyzer
Log Analyzer
Symptom Editor

Description

The IBM Monitoring and Diagnostic Tools for Java - Health Center is a

Restrictions

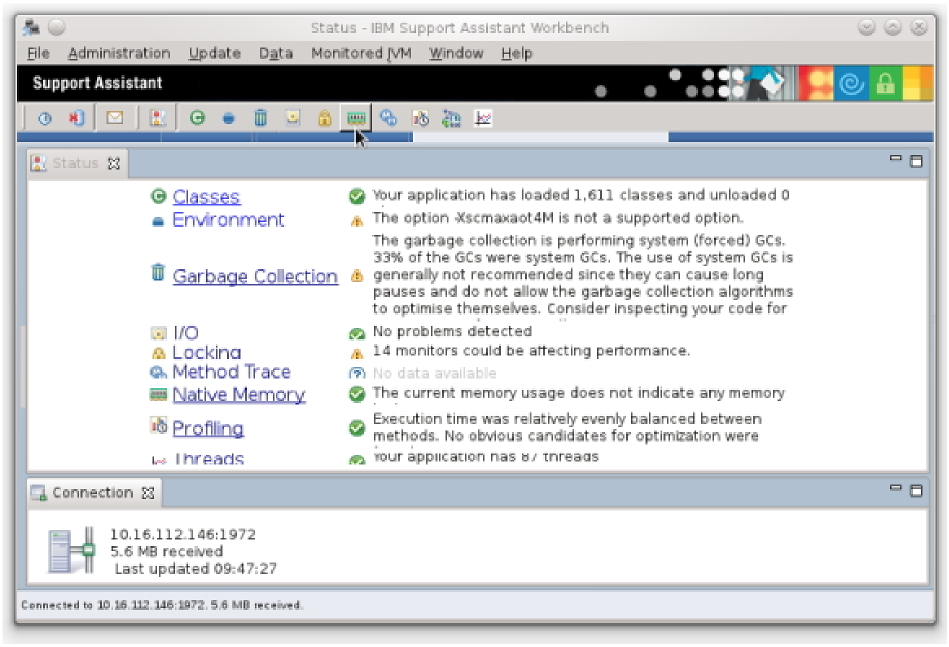
None

Associations

Tool is not associated with any products

4. A connection wizard will display. Click **Next**. Specify the Hostame or IP address where the application server is running. By default, port 1972 will be used. If you have any security requirements, specify them here, otherwise click **Next**. If the hostname and port are found, click **Next** again, otherwise figure out why the connection didn't work. If successful, Health Center will look something like Figure 11.

Figure 11. Health Center default view


Status - IBM Support Assistant Workbench
File Administration Update Data Monitored JVM Window Help
Support Assistant
Status
Connection

Classes — Your application has loaded 1,611 classes and unloaded 0
Environment — The option -Xscmaxaot4M is not a supported option.
Garbage Collection — The garbage collection is performing system (forced) GCs. 33% of the GCs were system GCs. The use of system GCs is generally not recommended since they can cause long pauses and do not allow the garbage collection algorithms to optimise themselves. Consider inspecting your code for
I/O — No problems detected
Locking — 14 monitors could be affecting performance.
Method Trace — No data available
Native Memory — The current memory usage does not indicate any memory
Profiling — Execution time was relatively evenly balanced between methods. No obvious candidates for optimization were
Threads — Your application has 87 threads

10.16.112.146:1972
5.6 MB received
Last updated 09:47:27

Connected to 10.16.112.146:1972. 5.6 MB received.

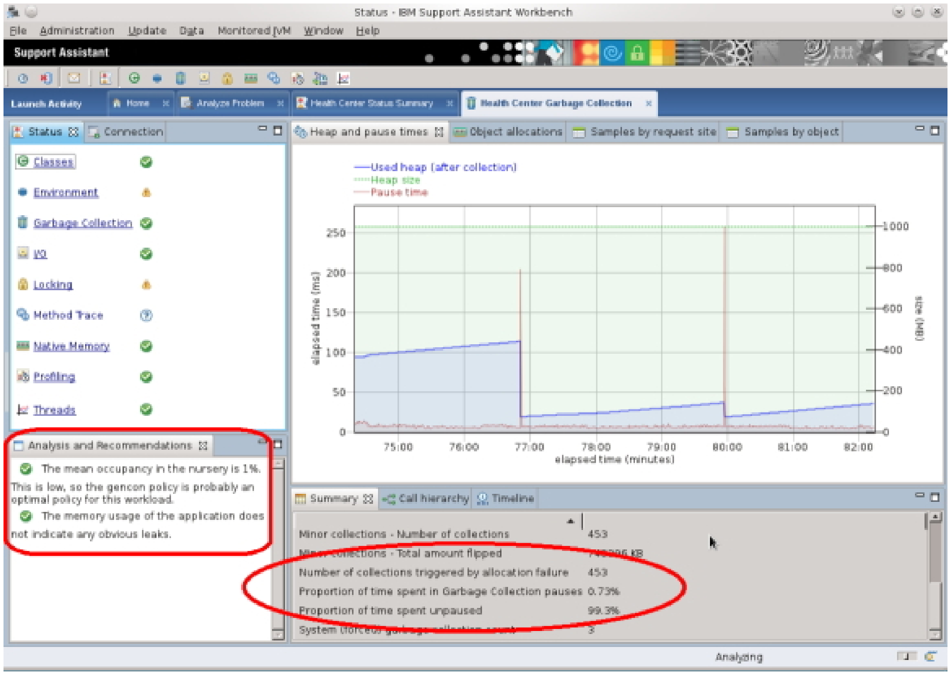
5. Maximize Health Center's screen and click around to get a feel for its features. At this point, you're ready to start some detailed performance analysis (which will be explored in the next sections. Go ahead and restart your JMeter load with the number of users found at your saturation point. Health Center will dynamically update as new information is available. Let the JMeter load run through your warm-up period before proceeding further.

Garbage collection analysis

The first step in any Java application performance analysis should always be studying the garbage collection statistics. With Health Center up and running, this is really easy to do.

Click on the **Garbage Collection** link in the Health Center window. A view similar to Figure 12 will display.

Figure 12. Health Center – Garbage Collection view



The mean occupancy in the nursery is 1%. This is low, so the gencon policy is probably an optimal policy for this workload.

The memory usage of the application does not indicate any obvious leaks.

Minor collections - Number of collections	453
Minor collections - Total amount flipped	[illegible] KB
Number of collections triggered by allocation failure	453
Proportion of time spent in Garbage Collection pauses	0.73%
Proportion of time spent unpaused	99.3%
System (forced) garbage collection count	3

There are two key things to review first for entry level analysis:

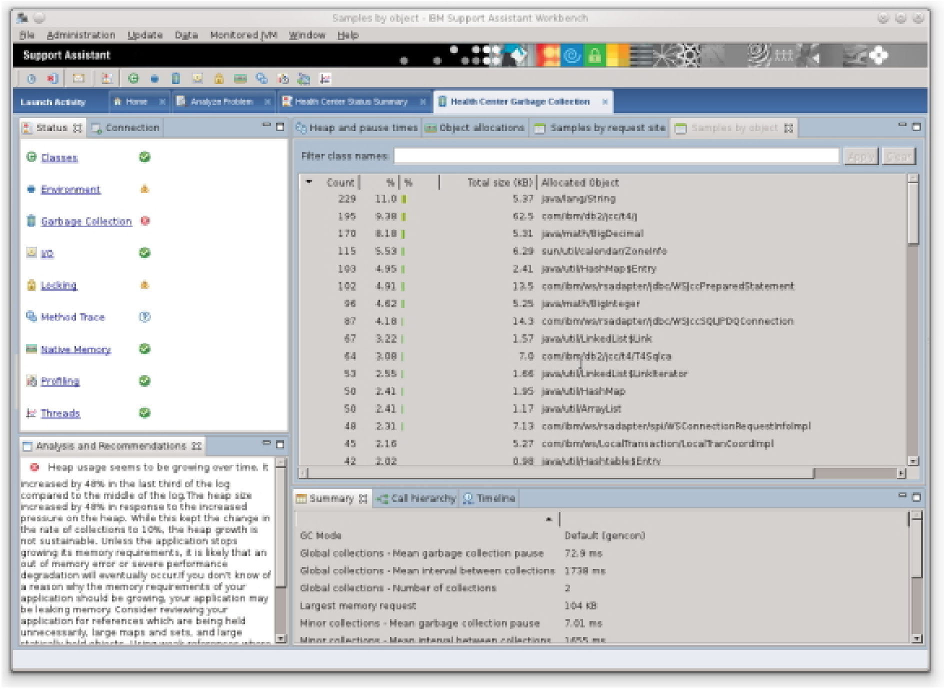
- The **Analysis and Recommendations** section in the bottom left corner provides useful tips and information based on built in intelligence in Health Center. These tips can indicate garbage collection policy and heap size recommendations, observations about memory leaks or System.gc() calls, and more. In Figure 12, this section tells you that gencon is an optimal GC policy for DayTrader, and that the application does not appear to be leaking memory. That's always a good starting point.
- The **Summary** panel at the bottom of the window contains data for the most important statistics that you should be concerned with, starting with the “Proportion of time spent in Garbage Collection pauses.” This single statistic tells you what percentage of the time your application is stopped because garbage collection is occurring. This number should be as low as possible, ideally less than 2-3%. If this number is 10%, then you could see as much as 10% higher throughput by optimal tuning to your JVM heap sizes and garbage collection policy. As mentioned before, this case study is an excellent article to help guide you through that tuning process.

A few other tips to help you find the data you are most interested in:

- The X-axis in the chart in Figure 12 shows the elapsed time since server startup. You can change this to chart against the time of day. This can be useful to correlate what activity was happening at certain times of the day. The X-axis can be changed by selecting the **context menu > Change Units > X-Axis > Time**.
- You can crop the data to eliminate the warm-up period to get a more accurate view of what's happening under the normal active conditions. To do so, select **Data > Crop Data** (to trim the start and finish) or just **Data > Reset Data** to clear any data up to this point.

- For more detailed analysis, click on the **Samples by object** tab. This enables you to see a breakdown of what objects are being allocated, how many are allocated of each type, and what the total size is. There's even an option to search by package or object name. An example is shown in Figure 13. Based on these results, it would probably be a good idea to review the application code to see if the usage of BigDecimal and BigInteger could be reduced.

Figure 13. Health Center – Samples By Object view



Count	%	%	Total size (KB)	Allocated Object
229	11.0		5.37	java/lang/String
195	9.38		62.5	com/ibm/db2/jcc/t4/j
170	8.18		5.31	java/math/BigDecimal
115	5.53		6.29	sun/util/calendar/ZoneInfo
103	4.95		2.41	java/util/HashMap$Entry
102	4.91		13.5	com/ibm/ws/rsadapter/jdbc/WSJccPreparedStatement
96	4.62		5.25	java/math/BigInteger
87	4.18		14.3	com/ibm/ws/rsadapter/jdbc/WSJccSQLJPDQConnection
67	3.22		1.57	java/util/LinkedList$Link
64	3.08		7.0	com/ibm/db2/jcc/t4/T4Sqlca
53	2.55		1.66	java/util/LinkedList$LinkIterator
50	2.41		1.95	java/util/HashMap
50	2.41		1.17	java/util/ArrayList
48	2.31		7.13	com/ibm/ws/rsadapter/spi/WSConnectionRequestInfoImpl
45	2.16		5.27	com/ibm/ws/LocalTransaction/LocalTranCoordImpl
42	2.02		0.98	java/util/Hashtable$Entry

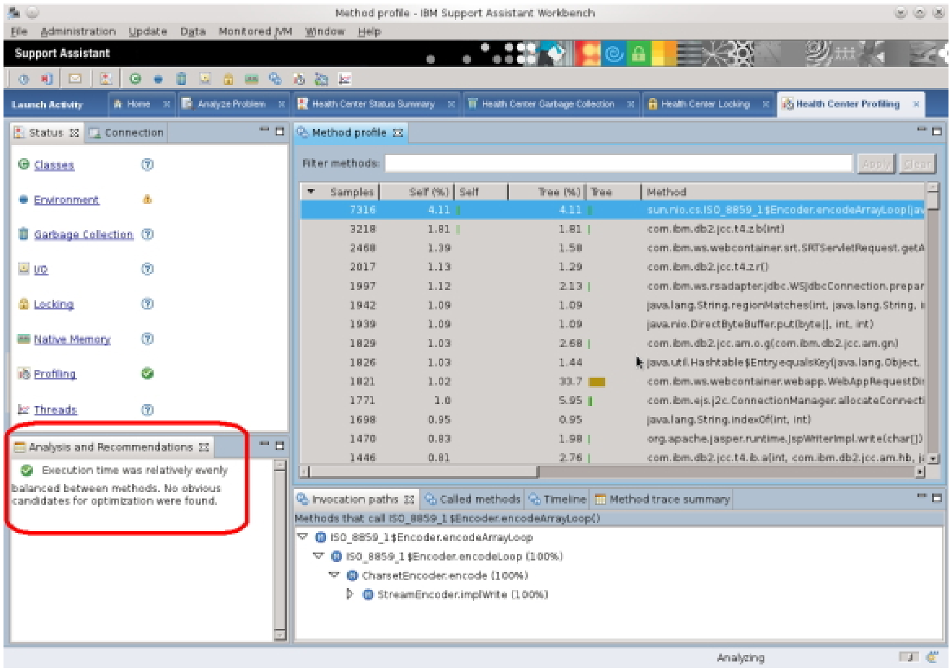
Heap usage seems to be growing over time. It increased by 48% in the last third of the log compared to the middle of the log.The heap size increased by 48% in response to the increased pressure on the heap. While this kept the change in the rate of collections to 10%, the heap growth is not sustainable. Unless the application stops growing its memory requirements, it is likely that an out of memory error or severe performance degradation will eventually occur.If you don't know of a reason why the memory requirements of your application should be growing, your application may be leaking memory. Consider reviewing your application for references which are being held unnecessarily, large maps and sets, and large

GC Mode	Default (gencon)
Global collections - Mean garbage collection pause	72.9 ms
Global collections - Mean interval between collections	1738 ms
Global collections - Number of collections	2
Largest memory request	104 KB
Minor collections - Mean garbage collection pause	7.01 ms
Minor collections - Mean interval between collections	1655 ms

Method profiling analysis

With the load driver still running, click on the **Profiling** link. This opens the Method Profiling view which looks like Figure 14.

Figure 14. Health Center – Method Profiling view



Execution time was relatively evenly balanced between methods. No obvious candidates for optimization were found.

Samples	Self (%)	Self	Tree (%)	Tree	Method
7316	4.11	▮	4.11	▮	sun.nio.cs.ISO_8859_1$Encoder.encodeArrayLoop(jav
3218	1.81	▮	1.81	▮	com.ibm.db2.jcc.t4.z.b(int)
2468	1.39		1.58		com.ibm.ws.webcontainer.srt.SRTServletRequest.getA
2017	1.13		1.29		com.ibm.db2.jcc.t4.z.r()
1997	1.12		2.13	▮	com.ibm.ws.rsadapter.jdbc.WSJdbcConnection.prepar
1942	1.09		1.09		java.lang.String.regionMatches(int, java.lang.String, i
1939	1.09		1.09		java.nio.DirectByteBuffer.put(byte[], int, int)
1829	1.03		2.68	▮	com.ibm.db2.jcc.am.o.g(com.ibm.db2.jcc.am.gn)
1826	1.03		1.44		java.util.Hashtable$Entry.equalsKey(java.lang.Object,
1821	1.02		33.7	▮	com.ibm.ws.webcontainer.webapp.WebAppRequestDi
1771	1.0		5.95	▮	com.ibm.ejs.j2c.ConnectionManager.allocateConnecti
1698	0.95		0.95		java.lang.String.indexOf(int, int)
1470	0.83		1.98	▮	org.apache.jasper.runtime.JspWriterImpl.write(char[])
1446	0.81		2.76	▮	com.ibm.db2.jcc.t4.ib.a(int, com.ibm.db2.jcc.am.hb, j

Methods that call ISO_8859_1$Encoder.encodeArrayLoop()

```
ISO_8859_1$Encoder.encodeArrayLoop
  ISO_8859_1$Encoder.encodeLoop (100%)
    CharsetEncoder.encode (100%)
      StreamEncoder.implWrite (100%)
```

The Method Profile table shows which methods are using the most processing resources. This is a view of the entire JVM, not just your particular application, so this will include method information for database drivers, WebSphere Application Server containers, and so on. It's helpful to look at this view to get the larger picture of what's going on in the server. As with before in the garbage collection analysis section, one of the best places to start is the Analysis and Recommendations section. This panel will highlight any methods that were found to be consuming a much larger portion of the CPU cycles than the rest. In the example above, the tip says that there are no obvious methods for optimizing since all the cycles are pretty evenly split. If a method or two were pointed out here, the code for that method should be reviewed to see if any optimizations can be made, or the number of times it is called can be reduced.

To dig deeper, you need to have an understanding of how to interpret the data in the table. To assist with that, refer to the Health Center documentation by selecting **Help > Help Contents**, then scroll down and expand the **Tool: IBM Monitoring and Diagnostic Tools for Java - Health Center** book. The documentation states:

Methods with a higher Self (%) value are described as "hot," and are good candidates for optimization. Small improvements to the efficiency of these methods might have a large effect on performance. You can optimize methods by reducing the amount of work that they do or by reducing the number of times that they are called. Methods near the end of the table are poor candidates for optimization. Even large improvements to their efficiency are unlikely to affect performance, because they do not use as much processing resource.

Here is a description of each of the columns in the table:

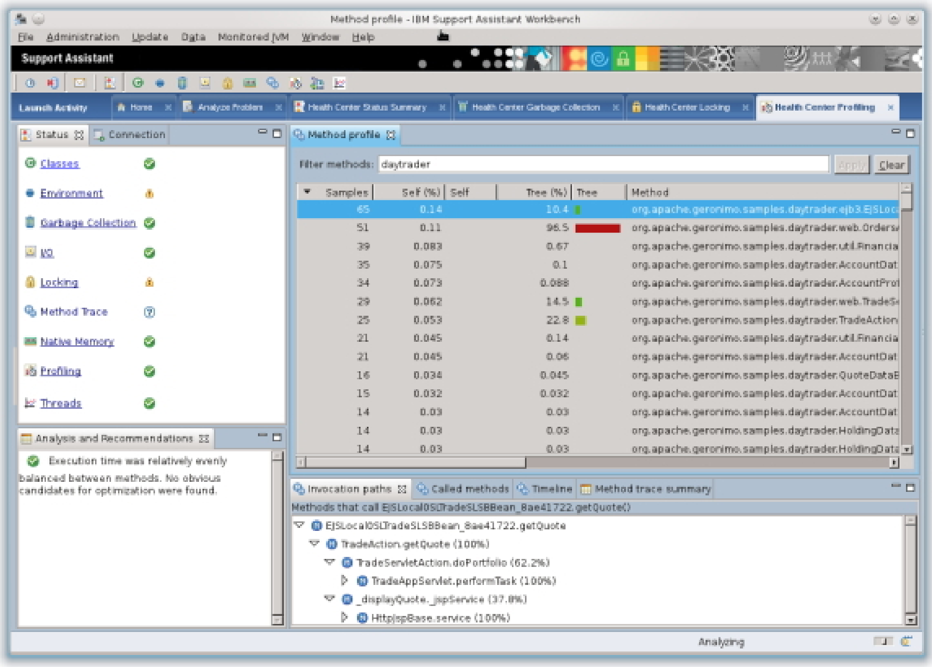
Table 1. Method profile table

Column	Description
Self (%)	The percentage of samples taken while a particular method was being run at the top of the stack. This value is a good indicator of how expensive a method is in terms of using processing resource.
Self	A graphical representation of the Self (%) column. Wider, redder bars indicate hotter methods.
Tree (%)	The percentage of samples taken while a particular method was anywhere in the call stack. This value shows the percentage of time that this method, and methods it called (descendants), were being processed. This value gives a good guide to the areas of your application where most processing time is spent.
Tree	A graphical representation of the Tree (%) column. Wider, redder bars indicate hotter method stacks.
Samples	The number of samples taken while a particular method was being run at the top of the stack.
Method	A fully qualified representation of the method, including package name, class name, method name, arguments, and return type.

Sort any of these columns by clicking the column header to sort in ascending or descending order. With this understanding, you can dig deep into the performance characteristics of your workload. Some useful tips for navigating through the table:

- Clicking on a row in the table will show you the full invocation path in the bottom panel of how the method got executed.
- The **Timeline** tab will show when the method was executed over the period for which the profiling has been active. This can be useful so you don't focus on something that was executed early on in the profile, perhaps during warm-up, but then goes away later on.
- The **Filter methods** text box is useful to search on specific classes and filter out the rest of profile. This should be used to drill down on details of your application only, to remove all the other non-application classes. As an example, Figure 15 shows the profile filtered on “daytrader” since all of the DayTrader application classes have “daytrader” in the package name. This enables you to focus on looking at the most resource intensive methods in just your application.

Figure 15. DayTrader filtered Method Profile view



Method profile - IBM Support Assistant Workbench

Status: Classes ✓, Environment, Garbage Collection ✓, I/O ✓, Locking, Method Trace, Native Memory ✓, Profiling ✓, Threads ✓

Analysis and Recommendations: Execution time was relatively evenly balanced between methods. No obvious candidates for optimization were found.

Filter methods: daytrader

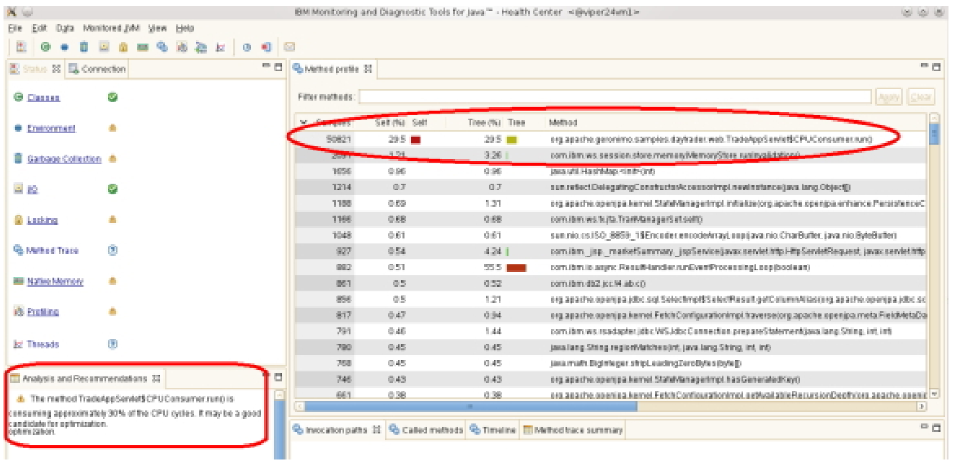
Samples	Self (%)	Self	Tree (%)	Tree	Method
65	0.14		10.4	🟩	org.apache.geronimo.samples.daytrader.ejb3.EJSLoc
51	0.11		96.5	🟥	org.apache.geronimo.samples.daytrader.web.Orders
39	0.083		0.67		org.apache.geronimo.samples.daytrader.util.Financia
35	0.075		0.1		org.apache.geronimo.samples.daytrader.AccountDat
34	0.073		0.088		org.apache.geronimo.samples.daytrader.AccountProf
29	0.062		14.5	🟩	org.apache.geronimo.samples.daytrader.web.TradeS
25	0.053		22.8	🟩	org.apache.geronimo.samples.daytrader.TradeAction
21	0.045		0.14		org.apache.geronimo.samples.daytrader.util.Financia
21	0.045		0.06		org.apache.geronimo.samples.daytrader.AccountDat
16	0.034		0.045		org.apache.geronimo.samples.daytrader.QuoteDataB
15	0.032		0.032		org.apache.geronimo.samples.daytrader.AccountDat
14	0.03		0.03		org.apache.geronimo.samples.daytrader.AccountDat
14	0.03		0.03		org.apache.geronimo.samples.daytrader.HoldingData
14	0.03		0.03		org.apache.geronimo.samples.daytrader.HoldingData

Methods that call EJSLocal0SLTradeSLSBBean_8ae41722.getQuote()

```
▽ EJSLocal0SLTradeSLSBBean_8ae41722.getQuote
  ▽ TradeAction.getQuote (100%)
    ▽ TradeServletAction.doPortfolio (62.2%)
      ▷ TradeAppServlet.performTask (100%)
    ▽ _displayQuote._jspService (37.8%)
      ▷ HttpJspBase.service (100%)
```

If the application has a particular method that is consuming a lot of the CPU cycles, then Health Center will flag it is a good candidate for optimization in the Analysis and Recommendations panel. An example of that is shown in Figure 16.

Figure 16. Optimization candidate example



IBM Monitoring and Diagnostic Tools for Java™ - Health Center <@viper24vm1>

Analysis and Recommendations: The method TradeAppServlet$CPUConsumer.run() is consuming approximately 30% of the CPU cycles. It may be a good candidate for optimization.

Samples	Self (%)	Self	Tree (%)	Tree	Method
50821	29.5	🟥	29.5	🟨	org.apache.geronimo.samples.daytrader.web.TradeAppServlet$CPUConsumer.run()
[illegible]	1.21		3.26		com.ibm.ws.session.store.memory.MemoryStore.runInvalidation()
1656	0.96		0.96		java.util.HashMap.<init>(int)
1214	0.7		0.7		sun.reflect.DelegatingConstructorAccessorImpl.newInstance(java.lang.Object[])
1188	0.69		1.31		org.apache.openjpa.kernel.StateManagerImpl.initialize(org.apache.openjpa.enhance.PersistenceC
1166	0.68		0.68		com.ibm.ws.tx.jta.TranManagerSet.self()
1048	0.61		0.61		sun.nio.cs.ISO_8859_1$Encoder.encodeArrayLoop(java.nio.CharBuffer, java.nio.ByteBuffer)
927	0.54		4.24	🟩	com.ibm._jsp._marketSummary._jspService(javax.servlet.http.HttpServletRequest, javax.servlet.http
882	0.51		55.5	🟥	com.ibm.io.async.ResultHandler.runEventProcessingLoop(boolean)
861	0.5		0.52		com.ibm.db2.jcc.t4.ab.c()
856	0.5		1.21		org.apache.openjpa.jdbc.sql.SelectImpl$SelectResult.getColumnAlias(org.apache.openjpa.jdbc.sc
817	0.47		0.94		org.apache.openjpa.kernel.FetchConfigurationImpl.traverse(org.apache.openjpa.meta.FieldMetaDa
791	0.46		1.44		com.ibm.ws.rsadapter.jdbc.WSJdbcConnection.prepareStatement(java.lang.String, int, int)
780	0.45		0.45		java.lang.String.regionMatches(int, java.lang.String, int, int)
768	0.45		0.45		java.math.BigInteger.stripLeadingZeroBytes(byte[])
746	0.43		0.43		org.apache.openjpa.kernel.StateManagerImpl.hasGeneratedKey()
661	0.38		0.38		org.apache.openjpa.kernel.FetchConfigurationImpl.setAvailableRecursionDepth(org.apache.openjpa

The Method Profiling view could be analyzed for days looking for performance improvements to an application. Since the output can be saved to a file, it makes comparing application changes extremely easy. Application developers could make a change that gets deployed and load tested with Health Center hooked in, and you can compare the previous profile information to see if the methods changed by the developers have increased or decreased in processing requirements.

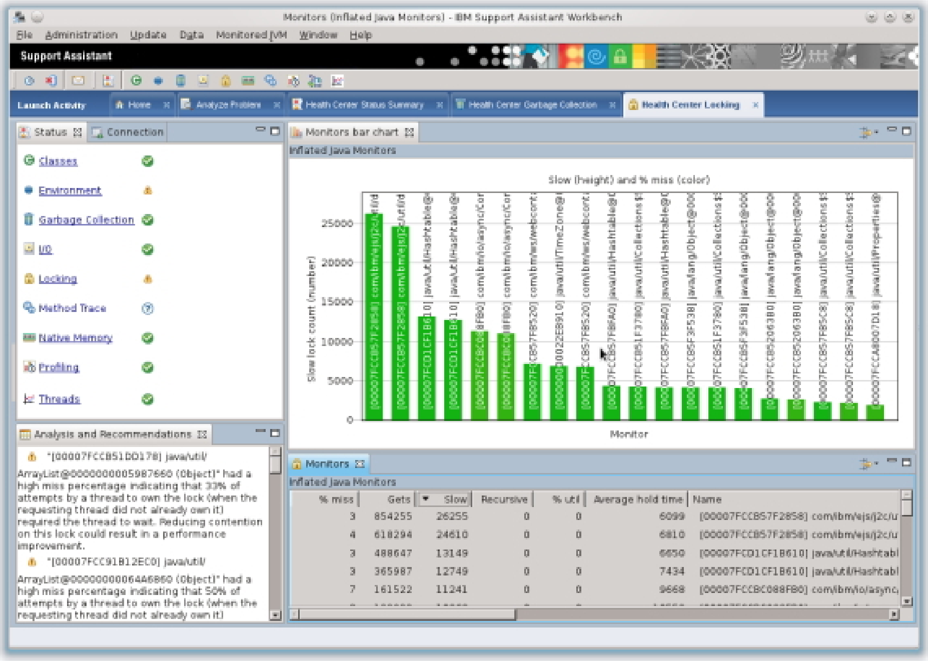
Locking analysis

Multi-threaded applications need to synchronize (or lock) shared resources to keep the state of the resource consistent. This consistency ensures that the status of one thread is not changed while another thread is reading it.

When locks are used in high-load applications that are deployed on systems with a large number of processors, the locking operation can prevent the application from using all the available processing resources. Imagine for a moment an application running on an 8-core machine with a major application code path being heavily synchronized such that only one thread can execute at a time. This could leave seven other threads waiting.

When running on large multi-core (4+) machines, lock analysis is essential to ensuring the application can scale up and utilize all the available hardware resources. (There's probably not much value in analysis here if the application is running on a single core machine.) The Locking perspective profiles lock usage and helps identify points of contention in the application or Java runtime that prevent the application from scaling. After clicking on the Locking link, a panel similar to Figure 17 should display.

Figure 17. Health Center – Locking view



"[00007FCCB51DD178] java/util/ArrayList@0000000005987660 (Object)" had a high miss percentage indicating that 33% of attempts by a thread to own the lock (when the requesting thread did not already own it) required the thread to wait. Reducing contention on this lock could result in a performance improvement.

"[00007FCC91B12EC0] java/util/ArrayList@00000000064A6860 (Object)" had a high miss percentage indicating that 50% of attempts by a thread to own the lock (when the requesting thread did not already own it)

Inflated Java Monitors

% miss	Gets	Slow	Recursive	% util	Average hold time	Name
3	854255	26255	0	0	6099	[00007FCCB57F2858] com/ibm/ejs/j2c/u
4	618294	24610	0	0	6810	[00007FCCB57F2858] com/ibm/ejs/j2c/u
3	488647	13149	0	0	6650	[00007FCD1CF18610] java/util/Hashtabl
3	365987	12749	0	0	7434	[00007FCD1CF18610] java/util/Hashtabl
7	161522	11241	0	0	9668	[00007FCCBC088FB0] com/ibm/io/async.

At first glance, the Monitors view can be overwhelming to try and understand. However, the Health Center documentation describes this panel in great detail to help you understand these metrics. Table 2 describes the contents of the columns in the table

Table 2. Monitors

Column	Description

% miss	The percentage of the total Gets, or acquires, for which the thread trying to enter the lock on the synchronized code had to block until it could take the lock.
Gets:	The total number of times the lock has been taken while it was inflated.
Slow:	The total number of non-recursive lock acquires for which the requesting thread had to wait for the lock because it was already owned by another thread.
Recursive:	The total number of recursive acquires. A recursive acquire occurs when the requesting thread already owns the monitor.
% util:	The amount of time the lock was held, divided by the amount of time the output was taken over.
Average hold time:	The average amount of time the lock was held, or owned, by a thread. For example, the amount of time spent in the synchronized block, measured in processor clock ticks.
Name:	The monitor name. This column is blank if the name is not known.

The table lists every Java monitor that was ever inflated. The % miss column is of initial interest. A high % miss shows that frequent contention occurs on the synchronized resource protected by the lock. This contention might be preventing the Java application from scaling further.

If a lock has a high % miss value, look at the average hold time and % util. Some tips:

- If % util and average hold time are both high, you might need to reduce the amount of work done while the lock is held.
- If % util is high but the average hold time is low, you might need to make the resource protected by the lock more granular to separate the lock into multiple locks.

Conclusion

Getting started in performance testing and analysis can be difficult at first without the right tools and knowledge. However, this article showed that there are some very simple steps that can be followed to ensure proper performance testing and that application bottlenecks have been removed such that it's performing as efficiently as possible.

Even though DayTrader might not resemble your application, the methodologies described in this paper for performance testing and identifying bottlenecks are the same. Testing with small user loads for slow periods up to high user loads for peak usage periods is vital to understanding your application's characteristics. Recording key metrics for comparisons as application or environment changes are made is essential to understanding where performance degradations might be coming from. Finally, the IBM Health Center tool makes performance analysis a breeze with garbage collection, method profiling, and lock profiling views to help you ensure the application is performing as efficiently as possible.

Related topics

- WebSphere Application Server Performance
- IBM Monitoring and Diagnostic Tools for Java - Health Center Version 2.0
- Tutorial: Hello World: Rational Performance Tester Tutorial
- Book: Performance Analysis for Java Websites
- How well does traditional performance testing apply to SOA solutions?
- The WebSphere Contrarian: Preparing for failure
- Case study: Tuning WebSphere Application Server V7 and V8 for performance
- The WebSphere Contrarian: Less might be more when tuning WebSphere Application Server