BEA WebLogic Server™

BEA WebLogic Server Performance and Tuning

Release 8.1 Beta
Document Revised: December 9, 2002

BEA WebLogic Server Performance and Tuning

Part Number	Document Revised	Software Version
N/A	December 9, 2002	BEA WebLogic Server Version 8.1 Beta

Contents

About This Document

To achieve the best performance for your WebLogic Server™ platform, you need to optimize the performance of the components that constitute the WebLogic Server environment. This document provides the following performance-related information:

- Chapter 1, “Tuning Hardware, Operating System, and Network Performance,” discusses hardware, operating system, and network performance issues.
- Chapter 2, “Tuning Java Virtual Machines (JVMs),” discusses JVM tuning considerations.
- Chapter 3, “Tuning WebLogic Server,” contains information on how to tune WebLogic Server to match your application needs.
- Chapter 4, “Tuning WebLogic Server EJBs,” describe how to tune WebLogic Server Enterprise Java Beans to match your application needs.
- Chapter 5, “Tuning WebLogic Server Applications,” discusses application tuning considerations.
- Appendix A, “Related Reading: Performance Tools and Information,” provides an extensive performance-related reading list.
- Appendix B, “Benchmark Tuning Examples for WebLogic Server,” provide recommendations for improving the out-of-the-box performance of WebLogic Server when running the ECPerf or SPECjAppServer 2001/2002 benchmarks.

The document also contains an index.

Audience

This document is written for people who monitor performance and tune the components in a WebLogic Server platform. It is assumed that readers know server administration and hardware performance tuning fundamentals, the WebLogic Server platform, XML, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation. Or you can go directly to the WebLogic Server Product Documentation page at http://edocs.bea.com/wls/docs81b.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site.

Related Information

For related information about administering and tuning WebLogic Server, see:

- *Configuring and Managing WebLogic Server* at http://edocs.bea.com/wls/docs81b/adminguide/index.html.
- BEA dev2dev Web site.
- The WebLogic Server performance “weblogic.developer.interest.performance” newsgroup available on the BEA Newsgroup server.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, and the title and document date of your documentation. If you have questions about this version of BEA WebLogic Server, or if you have problems installing and running it, contact BEA Customer Support through BEA WebSupport at http://www.bea.com, or by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String` *`CustomerName`*`;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

`[ ]`	Optional items in a syntax line. *Example*:
	`java utils.MulticastTest -n` *`name`* `-a` *`address`*
	`[-p` *`portnumber`*`] [-t` *`timeout`*`] [-s` *`send`*`]`

Convention	Usage

`|`	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`     password {application} {source}`

`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Tuning Hardware, Operating System, and Network Performance

The following sections describe issues related to optimizing hardware, operating system, and network performance:

- "Hardware Tuning" on page 1-1
- "Operating System Tuning" on page 1-3
- "Network Performance" on page 1-4

Hardware Tuning

When you examine performance, several factors influence how much capacity a given hardware configuration needs in order to support WebLogic Server™ and a given application.

Table 1-1 provides links to information about hardware tuning and standardized benchmarks and metrics.

Table 1-1 Platform-Specific Hardware Tuning Information

Issue	**For more information**
Supported Configurations pages	The Supported Configurations pages are frequently updated and contain the latest certification information on various platforms.
Solaris	For BEA WebLogic Server and Solaris-specific details, Sun Microsystems SPARC Solaris 2.5.1, 2.6, 2.7 and Sun Microsystems SPARC with Solaris 8 on the Supported Configurations pages. See also "Sun Microsystems Information" on page A-2.
Hewlett-Packard	For BEA WebLogic Server and HP-UX-specific details, see Hewlett-Packard HP/9000 with HP-UX 11.0 and 11i on the Certifications Pages. See also "Hewlett-Packard Company Information" on page A-3.
Standardized benchmarks and metrics	The Standard Performance Evaluation Corporation provides a set of standardized benchmarks and metrics for evaluating computer system performance.

Operating System Tuning

Tune your operating system according to your operating system documentation. BEA certifies WebLogic Server on multiple operating systems. Table 1-2 presents links to further information about operating system tuning.

Table 1-2 Operating System Considerations

Issue	For more information
Supported Configurations pages	The Supported Configurations pages are frequently updated and contains the latest certification information on various platforms.
File descriptors	On the UNIX platform, each socket connection to the server consumes a file descriptor. You need to configure your operating system to have the appropriate number of file descriptors. See "Tuning Solaris File Descriptor Limits" on the Sun Microsystems SPARC Solaris 2.5.1, 2.6, 2.7 and Sun Microsystems SPARC with Solaris 8 Supported Configurations pages.
Solaris TCP tuning parameters	See "Setting Solaris Tunable Parameters" on the Sun Microsystems SPARC Solaris 2.5.1, 2.6, 2.7 and Sun Microsystems SPARC with Solaris 8 Supported Configurations pages.
Maximum memory for a user process	Check your operating system documentation for the maximum memory available for a user process. In some operating systems, this value is as low as 128 MB. Also, refer to your operating system documentation. For more information about memory management, see Chapter 2, "Tuning Java Virtual Machines (JVMs)."

Table 1-2 Operating System Considerations (Continued)

Issue	For more information
Using native I/O for serving static files (Windows only)	When running WebLogic Server on Windows you can specify that WebLogic Server use the native operating system call `TransmitFile` instead of using Java methods to serve static files such as HTML files, text files, and image files. Using native I/O can provide performance improvements when serving larger static files. See "Using Native I/O for Serving Static Files" for more information.

Network Performance

When the supply of network resources is unable to keep up with the demand for resources, performance degrades. Table 1-3 presents some issues to consider.

Table 1-3 Network Configuration Considerations

Issue	Consideration
Network hardware and software	If you have a problem with one or more network components (hardware or software), work with your network administrator to isolate and eliminate the problem.
Network bandwidth	Make sure that you have an appropriate amount of network bandwidth available for WebLogic Server and the connections it makes to other tiers in your architecture, such as client and database connections. See "Determining Network Bandwidth" on page 1-5.

Table 1-3 Network Configuration Considerations (Continued)

Issue	Consideration
LAN infrastructure	Your local area network must be fast enough to handle your application's peak capacity. If you have high network traffic that consistently exceeds the capacity of the available resources (for example, the hit rate on a Web server has reached its maximum value while the system is 100 percent busy), do one of the following: ■ Redesign the network and redistribute the load ■ Reduce the number of network clients ■ Increase the number of systems handling the network load

Determining Network Bandwidth

A WebLogic Server machine requires enough bandwidth to handle all of its client connections. In the case of programmatic clients, each client JVM has a single socket to the server. Each socket requires bandwidth. A WebLogic Server handling programmatic clients should have 125–150 percent of the bandwidth that a similar Web server would handle. If you are handling only HTTP clients, expect a bandwidth requirement similar to a Web server serving static pages.

To determine the bandwidth required to run a Web server, you can assume that each 56kps of bandwidth can handle 7–10 simultaneous requests, depending upon the size of the content that you are delivering.

To determine whether you have enough bandwidth in a given deployment, use the network monitoring tools provided by your network operating system vendor to see what the load is on the network system. If the load is very high, bandwidth may be a bottleneck for your system.

CHAPTER

2 Tuning Java Virtual Machines (JVMs)

The Java virtual machine (JVM) is a virtual "execution engine" instance that executes the bytecodes in Java class files on a microprocessor. How you tune your JVM affects the performance of WebLogic Server and your applications.

The following sections discuss JVM tuning options for WebLogic Server:

- "JVM Tuning Considerations" on page 2-2
- "JVM Heap Size and Garbage Collection" on page 2-3
- "Specifying Heap Size Values" on page 2-7
- "Automatically Detecting Low Memory Conditions and Forcing Garbage Collection" on page 2-9
- "Manually Forcing Garbage Collection" on page 2-10
- "Setting Java HotSpot VM Options" on page 2-11

BEA WebLogic JRockit 8.1 Beta (running with JS2E version 1.4.1,) is developed uniquely for server-side applications and optimized for Intel architectures. JRockit is designed to ensure reliability, scalability, manageability, and flexibility for Java applications running on any kind of hardware architecture at significantly lower costs to the enterprise. For more information on using JRockit and supported platforms, see the *JRockit for Windows and Linux User Guide*.

For links to related reading for JVM tuning, see Appendix A, "Related Reading: Performance Tools and Information."

JVM Tuning Considerations

Table 2-1 presents general JVM tuning considerations.

Table 2-1 General JVM Tuning Considerations

Issue	Description
JVM vendor and version	Use only production JVMs on which WebLogic Server has been certified. WebLogic Server 8.1 supports only those JVMs that are Java 1.3-compliant. The Supported Configurations pages are frequently updated and contains the latest certification information on various platforms.
Tuning heap size and garbage collection	For WebLogic Server heap size tuning details, see "JVM Heap Size and Garbage Collection" on page 2-3. For a good overview of garbage collection on `java.sun.com`, see Tuning Garbage Collection with the 1.3.1 Java Virtual Machine.
Generational garbage collection	See "Generational Garbage Collection" on page 2-4.
Mixed client/server JVMs	Deployments using different JVM versions for the client and server are supported in WebLogic Server. For more information, see the support page for Mixed Client/Server JVMs.
UNIX threading models	There are two UNIX threading models: green threads and native threads. To get the best performance and scalability with WebLogic Server, choose a JVM that uses native threads. For Solaris, see "Threading Models and Solaris Versions Supported" on Sun Microsystems' Web site.

Table 2-1 General JVM Tuning Considerations (Continued)

Issue	Description
Just-in-Time (JIT) JVMs	Use a JIT compiler when you run WebLogic Server. Most JVMs use a JIT compiler, including those from Sun Microsystems and Symantec. See your JVM supplier documentation for more information. **Note:** The Sun Microsystems' JVM 1.3.x, JIT options are no longer valid. See "Java Virtual Machine (JVM) Information" on page A-5.

JVM Heap Size and Garbage Collection

Garbage collection is the JVM's process of freeing up unused Java objects in the Java heap.The Java heap is where the objects of a Java program live. It is a repository for live objects, dead objects, and free memory. When an object can no longer be reached from any pointer in the running program, it is considered "garbage" and ready for collection.

The JVM heap size determines how often and how long the VM spends collecting garbage. An acceptable rate for garbage collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections. If you set a large heap size, full garbage collection is slower, but it occurs less frequently. If you set your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently.

The goal of tuning your heap size is to minimize the time that your JVM spends doing garbage collection while maximizing the number of clients that WebLogic Server can handle at a given time. To ensure maximum performance during benchmarking, you might set high heap size values to ensure that garbage collection does not occur during the entire run of the benchmark.

You might see the following Java error if you are running out of heap space:

```
java.lang.OutOfMemoryError <<no stack trace available>>
java.lang.OutOfMemoryError <<no stack trace available>>
Exception in thread "main"
```

To modify heap space values, see “Specifying Heap Size Values” on page 2-7.

To configure WebLogic Server to automatically detect when you are running out of heap space and to address low memory conditions in the server, see “Automatically Detecting Low Memory Conditions and Forcing Garbage Collection” on page 2-9.

Generational Garbage Collection

The 1.3 Java HotSpot JVM uses a *generational* collector that provides significant increases in allocation speed and overall garbage collection efficiency. While *naive* garbage collection examines every reachable object in the heap, generational garbage collection considers the lifetime of an object to avoid extra collection work. The Hotspot JVM operates on the assumption that a majority of objects die young, and do not need to be considered for collection, which makes for efficient garbage collection.

With generational garbage collection, the Java heap is divided into two general areas: Young and Old. The Young generation area is subdivided further into Eden and two survivor spaces. Eden is the area where new objects are allocated. When garbage collection occurs, live objects in Eden are copied into the next survivor space. Objects are copied between survivor spaces in this way until they exceed a maximum heap size threshold, and then they are moved out of the Young area and into the Old. For information about specifying the size and ratios of the Young and Old generation areas, see “Specifying Heap Size Values” on page 2-7.

Many objects become garbage shortly after being allocated. These objects are said to have “infant mortality.” The longer an object survives, the more garbage collection it goes through, and the slower garbage collection becomes. The rate at which your application creates and releases objects affects the heap size, which in turn determines how often garbage collection occurs. Therefore, attempt to cache objects for re-use, whenever possible, rather than creating new objects.

Knowing that a majority of objects die young allows you to tune for efficient garbage collection. When you manage memory in generations, you create memory pools to hold objects of different ages. Garbage collection can occur in each generation when it fills up. If you can arrange for most of your objects to survive less than one collection, garbage collection is very efficient. Poorly sized generations cause frequent garbage collection, which can affect performance.

For a good overview of generational garbage collection, see Tuning Garbage Collection with the 1.3.1 Java Virtual Machine

Using Verbose Garbage Collection to Determine Heap Size

Verbose garbage collection (`verbosegc`) enables you to measure exactly how much time and resources are put into garbage collection. To determine the most effective heap size, turn on verbose garbage collection and redirect the output to a log file for diagnostic purposes.

The following steps outline this procedure:

1. Monitor the performance of WebLogic Server under maximum load while running your application.
2. Use the `-verbosegc` option to turn on verbose garbage collection output for your JVM and redirect *both* the standard error and standard output to a log file.

   This places thread dump information in the proper context with WebLogic Server informational and error messages, and provides a more useful log for diagnostic purposes.

   For example, on Windows and Solaris, enter the following:

   ```
   % java -ms32m -mx200m -verbosegc -classpath $CLASSPATH
   -Dweblogic.Name=%SERVER_NAME% -Dbea.home="C:\bea"
   -Dweblogic.management.username=%WLS_USER%
   -Dweblogic.management.password=%WLS_PW%
   -Dweblogic.management.server=%ADMIN_URL%
   -Dweblogic.ProductionModeEnabled=%STARTMODE%
   -Djava.security.policy="%WL_HOME%\server\lib\weblogic.policy"
   weblogic.Server
   >> logfile.txt 2>&1
   ```

   where the `logfile.txt 2>&1` command redirects both the standard error and standard output to a log file.

   On HPUX, use the following option to redirect `stderr stdout` to a single file:

   ```
      -Xverbosegc:file=/tmp/gc$$.out
   ```

   where `$$` maps to the process ID (PID) of the Java process. Because the output includes timestamps for when garbage collection ran, you can infer how often garbage collection occurs.
3. Analyze the following data points:

   a. How often is garbage collection taking place? In the `weblogic.log` file, compare the time stamps around the garbage collection.

   b. How long is garbage collection taking? Full garbage collection should not take longer than 3 to 5 seconds.

   c. What is your average memory footprint? In other words, what does the heap settle back down to after each full garbage collection? If the heap always settles to 85 percent free, you might set the heap size smaller.

4. If you are using 1.3 Java HotSpot JVM, set the New generation heap sizes.

   See "Specifying Heap Size Values" on page 2-7 and Table 2-2, "Java Heap Size Options," on page 2-8.

5. Make sure that the heap size is not larger than the available free RAM on your system.

   Use as large a heap size as possible without causing your system to "swap" pages to disk. The amount of free RAM on your system depends on your hardware configuration and the memory requirements of running processes on your machine. See your system administrator for help in determining the amount of free RAM on your system.

6. If you find that your system is spending too much time collecting garbage (your allocated "virtual" memory is more than your RAM can handle), lower your heap size.

   Typically, you should use 80 percent of the available RAM (not taken by the operating system or other processes) for your JVM.

7. If you find that you have a large amount of available free RAM remaining, run more instances of WebLogic Servers on your machine.

   Remember, the goal of tuning your heap size is to minimize the time that your JVM spends doing garbage collection while maximizing the number of clients that WebLogic Server can handle at a given time.

Specifying Heap Size Values

Java heap size values must be specified whenever you start WebLogic Server. This can be done either from the Java command line or by modifying the default values in the sample startup scripts that are provided with the WebLogic distribution for starting WebLogic Server.

For example, when starting WebLogic Server from a Java command line, the heap size values could be specified as follows:

```
$ java -XX:NewSize=128m -XX:MaxNewSize=128m -XX:SurvivorRatio=8
-Xms512m -Xmx512m
-Dweblogic.Name=%SERVER_NAME% -Dbea.home="C:\bea"
-Dweblogic.management.username=%WLS_USER%
-Dweblogic.management.password=%WLS_PW%
-Dweblogic.management.server=%ADMIN_URL%
-Dweblogic.ProductionModeEnabled=%STARTMODE%
-Djava.security.policy="%WL_HOME%\server\lib\weblogic.policy"
 weblogic.Server
```

The default size for these values is measured in bytes. Append the letter ‘k’ or ‘K’ to the value to indicate kilobytes, ‘m’ or ‘M’ to indicate megabytes, and ‘g’ or ‘G’ to indicate gigabytes. For more information on the heap size options, see “Java Heap Size Options” on page 2-8.

Using WebLogic Startup Scripts to Set Heap Size

Sample startup scripts are provided with the WebLogic Server distribution for starting the server and for setting the environment to build and run the server:

- `startWLS.cmd` and `setEnv.cmd` for Windows systems.
- `startWLS.sh` and `setEnv.sh` for UNIX systems.

These scripts are located in *`WL_HOME`*`\server\bin`, where *`WL_HOME`* is the location in which you installed WebLogic Server. The startup scripts set environment variables, such as the default memory arguments passed to Java (that is, heap size) and the location of the JDK, and then starts the JVM with WebLogic Server arguments.

Be aware that the WebLogic Server startup scripts specify default heap size parameters; therefore, you will need to modify them to fit your environment and applications. See "Starting an Administration Server Using a Script" .

Java Heap Size Options

You achieve best performance by individually tuning each of your applications. However, configuring the JVM heap size options listed in Table 2-2 when starting WebLogic Server increases performance for most applications.

These options may differ depending on your architecture and operating system. See your vendor's documentation for platform-specific JVM tuning options.

Table 2-2 Java Heap Size Options

Task	Option	Description
Setting the New generation heap size	`-XX:NewSize`	Use this option to set the New generation Java heap size. Set this value to a multiple of 1024 that is greater than 1MB. As a general rule, set `-XX:NewSize` to be one-fourth the size of the maximum heap size. Increase the value of this option for larger numbers of short-lived objects. Be sure to increase the New generation as you increase the number of processors. Memory allocation can be parallel, but garbage collection is not parallel.
Setting the maximum New generation heap size	`-XX:MaxNewSize`	Use this option to set the maximum New generation Java heap size. Set this value to a multiple of 1024 that is greater than 1MB.
Setting New heap size ratios	`-XX:SurvivorRatio`	The New generation area is divided into three sub-areas: Eden, and two survivor spaces that are equal in size. Use the `-XX:SurvivorRatio=X` option to configure the ratio of the Eden/survivor space size. Try setting this value to 8, and then monitor your garbage collection.

Table 2-2 Java Heap Size Options (Continued)

Task	Option	Description
Setting minimum heap size	`-Xms`	Use this option to set the minimum size of the memory allocation pool. Set this value to a multiple of 1024 that is greater than 1MB. As a general rule, set minimum heap size (`-Xms`) equal to the maximum heap size (`-Xmx`) to minimize garbage collections.
Setting maximum heap size	`-Xmx`	Use this option to set the maximum Java heap size. Set this value to a multiple of 1024 that is greater than 1MB.

Automatically Detecting Low Memory Conditions and Forcing Garbage Collection

WebLogic Server enables you to automatically detect and address low memory conditions in the server, and forces garbage collection if necessary. WebLogic Server detects low memory by sampling the available free memory a set number of times during a time interval. At the end of each interval, an average of the free memory is recorded and compared to the average obtained at the next interval. If the average drops by a user-configured amount after any sample interval, the server logs a low memory warning message in the log file and sets the server health state to “warning.”

If the average free memory ever drops below 5 percent of the initial free memory recorded immediately after you start the server, WebLogic Server automatically forces garbage collection and logs a message to the log file.

You configure each aspect of the low memory detection process using the Administration Console:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the navigation tree to display the servers configured in your domain.

4. Click the name of the server instance that you want to configure. Note that you configure low memory detection on a per-server basis.
5. Select the Configuration →Memory tab in the right pane.
6. Modify the following attributes as necessary to tune low memory detection for the selected server instance:
   - `Low Memory GCThreshold`: Enter a percentage value (0–9 percent) to represent the threshold after which WebLogic Server performs automatic garbage collection. By default, `Memory GCThreshold` is set to 5 percent. This means that the server forces garbage collection after the average free memory reaches 5 percent of the initial free memory measured at the server's boot time.
   - `Low Memory Granularity Level`: Enter a percentage value (1–99 percent) to use for logging low memory conditions and changing the server health state to "warning." By default this value is set to 5 percent. This means that if the average free memory drops by 5 percent or more over two measured intervals, the server logs a low memory warning in the log file and changes the server health state to "warning."
   - `Low Memory Sample Size`: Enter the number of times the server samples free memory during a fixed time period. By default, the server samples free memory 10 times each interval to acquire the average free memory. Using a higher sample size can increase the accuracy of the reading.
   - `Low Memory Time Interval`: Enter the time, in seconds, that define the interval over which the server determines average free memory values. By default WebLogic Server obtains an average free memory value every 3600 seconds.
7. Click Apply to apply your changes.
8. Reboot the server to use the new low memory detection attributes.

Manually Forcing Garbage Collection

Make sure that full garbage collection is necessary before forcing it on a server. When you force garbage collection, the JVM often examines every living object in the heap.

To use the Administration Console to force garbage collection on a specific server instance:

1. On the Administration Console, click the server instance node in the navigation tree for the server whose memory usage you want to view. A dialog box in the right pane shows the tabs associated with this instance.
2. Select the Monitoring →Performance tab.
3. Check the Memory Usage graph for high usage. Note that the Memory Usage graph displays information only for a server that is currently running.
4. Click the Force garbage collection button to force garbage collection. A message indicates that the collection operation was successful.

Setting Java HotSpot VM Options

You can use standard `java` command-line options to improve the performance of your JVM. How you use these options depends on how your application is coded. Although command-line options are consistent across platforms, some platforms may have different defaults.

Test both your client and server JVMs to see which options perform better for your particular application. The Sun Microsystems Java HotSpot VM Options document provides information on the command-line options and environment variables that can affect the performance characteristics of the Java HotSpot Virtual Machine.

See "Non-Standard Java Options for Windows and UNIX" on page 2-13 for more VM options that affect performance.

Standard Java Options for Windows and UNIX

In Windows, WebLogic Server invokes a particular version of the JVM through the `java` command and by specifying one of the options listed in Table 2-3.

Table 2-3 Standard Options for HotSpot VM on Windows

Option	Description
`-hotspot`	Selects the HotSpot Client VM, which according to Sun Microsystems, “provides superior performance to that of the Classic VM.”
`-classic`	Selects the Classic VM, which is essentially the same virtual machine implementation as in version 1.2 of the Java 2 SDK. **Note:** The Java 2 Classic VM is included only in the Java 2 SDK. It is not included in the Java 2 Runtime Environment. The `-classic` option will not work with the Java 2 Runtime Environment.

In UNIX, the WebLogic Server invokes a particular version of the JVM through the `java` command and by specifying one of the options listed in Table 2-4.

Table 2-4 Standard Options for HotSpot VM on UNIX

Option	Description
`-client or -hotspot`	Selects the HotSpot Client VM.
`-server`	Selects the HotSpot Server VM.

The Sun Microsystems The Java HotSpot Client and Server Virtual Machines document discusses the two implementations of the Java virtual machine that are available for J2SE 1.3.

Non-Standard Java Options for Windows and UNIX

You can also use non-standard `java` options to improve performance. How you use these options depends on how your application is coded. Although command-line options are consistent across platforms, some platforms may have different defaults. Note that non-standard command-line options are subject to change in future releases.

Two examples of non-standard options for improving performance on the Hotspot VM on Windows are listed in Table 2-5.

Table 2-5 Non-standard Options for HotSpot VM on Windows

Option	Description
`-Xnoclassgc`	This option disables garbage collection for the specified class. It prevents reloading of the class when the class is referenced after all references to it have been lost. This option requires an increased heap size.
`-Xrs`	Reduces usage of operating-system signals by the JVM.

For additional examples of non-standard Windows options, see Non-Standard Options (for Windows VMs).

Two examples of non-standard options for improving performance on the Hotspot VM on UNIX Solaris are listed in Table 2-6.

Table 2-6 Non-standard Options for HotSpot VM on Solaris

Option	Description
`-Xnoclassgc`	This option disables garbage collection for the specified class. It prevents reloading of the class when the class is referenced after all references to it have been lost. This option requires an increased heap size.
`-ss`	This option controls the native thread stack size. Setting it too high (>2MB) severely degrades performance.

For more examples of non-standard options for Solaris, see Non-Standard Options (for Solaris VMs).

CHAPTER

3 Tuning WebLogic Server

The following sections describe how to tune WebLogic Server to match your application needs.

Setting Performance-Related config.xml Parameters

The WebLogic Server configuration file (`config.xml`) contains a number of performance-related parameters that can be fine-tuned depending on your environment and applications. The `config.xml` file, located on the machine that hosts the Administration Server, provides persistent storage of Mbean attribute values. Every time you change an attribute using the system administration tools (using either the command-line interface or the Administration Console), its value is stored in the appropriate administration MBean and written to the `config.xml` file. Each WebLogic Server domain has its own `config.xml` file.

For more information about using WebLogic Server system administration tools, see the "System Administration Tools" section in the *Administration Guide*.

Table 3-1 lists the `config.xml` file parameters that affect server performance.

Table 3-1 Performance-Related config.xml Elements

Element	Attributes	For information
`Server`	`NativeIOEnabled`	See "Using WebLogic Server Performance Packs" on page 3-3.
`ExecuteQueue`	`ThreadCount`	See "Setting Thread Count" on page 3-4.

`ExecuteQueue`	`QueueLength`
	`QueueLengthThresholdPercent`
	`ThreadsIncrease`
	`ThreadsMaximum`
	`ThreadsMinimum`
		See "Tuning Execute Queues for Overflow Conditions" on page 3-9.

`Server`	`StuckThreadMaxTime` `StuckThreadTimerInterval`	See "Detecting "Stuck" Threads" on page 3-12.
`Server`	`ThreadPoolPercentSocketReaders`	See "Allocating Threads to Act as Socket Readers" on page 3-8.
`Server`	`AcceptBacklog`	See "Tuning Connection Backlog Buffering" on page 3-13.
`JDBCConnectionPool`	`InitialCapacity` `MaxCapacity`	See "How JDBC Connection Pools Enhance Performance" on page 3-14.
`JDBCConnectionPool`	`PreparedStatementCacheSize`	See "Caching Prepared and Callable Statements" on page 3-15.

Using WebLogic Server Performance Packs

Benchmarks show major performance improvements when you use native performance packs on machines that host WebLogic Server instances. Performance packs use a platform-optimized, native socket multiplexor to improve server performance. However, if you must use the pure-Java socket reader implementation for host machines, you can still improve the performance of socket communication by configuring the proper number of socket reader threads for each server instance and client machine.

Which Platforms Have Performance Packs?

To see which platforms currently have performance packs available:

1. Go to the Certifications Pages.
2. Select your platform from the list of certified platforms.
3. Use your browser's Edit →Find to locate all instances of "Performance Pack" to verify whether it is included for the platform.

Enabling Performance Packs

Make sure the `NativeIOEnabled` attribute of the `Server` element is defined in your `config.xml` file. The default `config.xml` file shipped with your distribution enables this attribute by default: `NativeIOEnabled=true`.

To use the Administration Console to make sure performance packs are enabled:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server instance that you want to configure.
5. Select the Configuration →Tuning tab.
6. If the Native IO Enabled check box is not selected, select the check box.

7. Click Apply.
8. Restart your server.

Setting Thread Count

The value of the `ThreadCount` attribute of an `ExecuteQueue` element in the `config.xml` file equals the number of simultaneous operations that can be performed by applications that use the execute queue. As work enters an instance of WebLogic Server, it is placed in an execute queue. This work is then assigned to a thread that does the work on it. Threads consume resources, so handle this attribute with care—you can degrade performance by increasing the value unnecessarily.

By default, a new WebLogic Server instance is configured with a default execute queue (named “default”) that contains 15 threads, which are used by all applications running on the server instance. A WebLogic Server instances also contains two built-in execute queues named `__weblogic_admin_html_queue` (on Administration Servers only) and `__weblogic_admin_rmi_queue` (on both Adminstration Servers and Managed Servers, but these queues are reserved for administrative communications. If you configure no additional execute queues, all Web applications and RMI objects use the default queue.

Note: If native performance packs are not being used for your platform, you may need to tune the default number of execute queue threads *and* the percentage of threads that act as socket readers to achieve optimal performance. For more information, see “Allocating Threads to Act as Socket Readers” on page 3-8.

Should You Modify the Default Thread Count?

Adding more threads to the default execute queue does not necessarily imply that you can process more work. Even if you add more threads, you are still limited by the power of your processor. Because threads consume memory, you can degrade performance by increasing the value of the `ThreadCount` attribute unnecessarily. A high execute thread count causes more memory to be used and increases context switching, which can degrade performance.

The value of the `ThreadCount` attribute depends very much on the type of work your application does. For example, if your client application is thin and does a lot of its work through remote invocation, that client application will spend more time connected — and thus will require a higher thread count — than a client application that does a lot of client-side processing.

If you do not need to use more than 15 threads (the default) for your work, do not change the value of this attribute. As a general rule, if your application makes database calls that take a long time to return, you will need more execute threads than an application that makes calls that are short and turn over very rapidly. For the latter case, using a smaller number of execute threads could improve performance.

Default Thread Count Scenarios

To determine the ideal thread count for an execute queue, monitor the queue's throughput while all applications in the queue are operating at maximum load. Increase the number of threads in the queue and repeat the load test until you reach the optimal throughput for the queue. (At some point, increasing the number of threads will lead to enough context switching that the throughput for the queue begins to decrease.)

Note: The WebLogic Server Administration Console displays the cumulative throughput for all of a server's execute queues. To access this throughput value, follow steps 1-6 in "Modifying the Thread Count in the Default Execute Queue" on page 3-7.

Table 3-2 shows default scenarios for adjusting available threads in relation to the number of CPUs on the WebLogic Server system. These scenarios also assume that the WebLogic Server is running under maximum load, and that all thread requests are satisfied by using the default execute queue. If you configure additional execute queues and assign applications to specific queues, monitor results on a pool-by-pool basis.

Table 3-2 Default Thread Count Scenarios

When...	Results	Do This:
Thread Count < number of CPUs	Your thread count is too low if: ■ CPU is waiting to do work, but there is work that could be done. ■ Cannot get 100 percent CPU utilization rate.	Increase the thread count.
Thread Count = number of CPUs	Theoretically ideal, but the CPUs are still under-utilized.	Increase the thread count.
Thread Count > number of CPUs (by a moderate number of threads)	Practically ideal, with a moderate amount of context switching and a high CPU utilization rate.	Tune the moderate number of threads and compare performance results.

Table 3-2 Default Thread Count Scenarios (Continued)

When...	Results	Do This:
Thread Count > number of CPUs (by a large number of threads)	Too much context switching, which can lead to significant performance degradation. Your performance may increase as you decrease the number of threads	Reduce the number of threads so that it equals the number of CPUs, and then add *only* the number of "stuck" threads that you have determined. For example, if you have four processors, then four threads can be running concurrently with the number of stuck threads. So, you want the execute threads to be 4 + the number of stuck threads. To determine the amount of stuck threads, see "Detecting "Stuck" Threads" on page 3-12. **Note:** This recommendation is highly application-dependent. For instance, the length of time the application might block threads can invalidate the formula.

Modifying the Thread Count in the Default Execute Queue

To modify the default execute queue thread count using the Administration Console:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server instance that contains the execute queue you want to configure. Note that you can only modify the default execute queue for the server, or a user-defined execute queue.
5. Select the Monitoring →General tab in the right pane.

6. Click the Monitor All Active Queues text link to display the execute queues that the selected server uses.
7. Click the Configure Execute Queue text link to display the execute queues that you can modify.
8. In the table of configured execute queues, click the name of the default execute queue to display the Execute Queue Configuration tab.
9. Increase or decrease, as appropriate, the default Thread Count value.
10. Click Apply to apply your changes.
11. Reboot the selected server to enable the new execute queue settings.

Assigning Applications to Execute Queues

Although you can configure the default execute queue to supply the optimal number threads for all WebLogic Server applications, configuring multiple execute queues can provide additional control for key applications. By using multiple execute queues, you can guarantee that selected applications have access to a fixed number of execute threads, regardless of the load on WebLogic Server. See "Using Execute Queues to Control Thread Usage" on page 5-4 for more information on assigning applications to configured execute queues.

Allocating Threads to Act as Socket Readers

For best socket performance, BEA recommends that you use the native socket reader implementation, rather than the pure-Java implementation, on machines that host WebLogic Server instances (see "Using WebLogic Server Performance Packs" on page 3-3). However, if you must use the pure-Java socket reader implementation for host machines, you can still improve the performance of socket communication by configuring the proper number of execute threads to act as socket reader threads for each server instance and client machine.

The `ThreadPoolPercentSocketReaders` attribute sets the maximum percentage of execute threads that are set to read messages from a socket. The optimal value for this attribute is application-specific. The default value is 33, and the valid range is 1–99.

Allocating execute threads to act as socket reader threads increases the speed and the ability of the server to accept client requests. It is essential to balance the number of execute threads that are devoted to reading messages from a socket and those threads that perform the actual execution of tasks in the server.

Set the Number of Socket Reader Threads on a WebLogic Server

To use the Administration Console to set the maximum percentage of execute threads that read messages from a socket:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server that you want to configure.
5. Select the Configuration →Tuning tab.
6. Edit the percentage of Java reader threads in the Socket Readers attribute field. The number of Java socket readers is computed as a percentage of the number of total execute threads (as shown in the Execute Threads attribute field).
7. Apply the changes.

Set the Number of Socket Reader Threads on Client Machines

On client machines, you can configure the number socket reader threads in the Java Virtual Machine (JVM) that runs the client. Specify the socket readers by defining the `-Dweblogic.ThreadPoolSize=`*`value`* and `-Dweblogic.ThreadPoolPercentSocketReaders=`*`value`* options in the `java` command line for the client.

Tuning Execute Queues for Overflow Conditions

You can configure a server to detect and optionally address potential overflow conditions in the default execute queue or any user-defined execute queue. WebLogic Server considers a queue to have a possible overflow condition when its current size reaches a user-defined percentage of its maximum size. When this threshold is reached, the server changes its health state to "warning" and can optionally allocate additional threads to perform the outstanding work in the queue, thereby reducing its size.

To automatically detect and address overflow conditions in a queue, you configure the following items:

- The threshold at which the server indicates an overflow condition. This value is set as a percentage of the configured size of the execute queue (the `QueueLength` value).
- The number of threads to add to the execute queue when an overflow condition is detected. These additional threads work to reduce the size of the queue and reduce the size of the queue to its normal operating size.
- The minimum and maximum number of threads available to the queue. In particular, setting the maximum number of threads prevents the server from assigning an overly high thread count in response to overload conditions.

To tune an execute queue using the WebLogic Server Administration Console:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server instance that contains the execute queue you want to configure. Note that you can only modify the default execute queue for the server, or a user-defined execute queue.
5. Select the Monitoring →General tab in the right pane.
6. Click the Monitor All Active Queues text link to display the execute queues that the selected server uses.
7. Click the Configure Execute Queue text link to display the execute queues that you can modify.
8. Click the name of the default execute queue or the user-defined execute queue that you want to configure to display the Execute Queue Configuration tab.

9. To specify how this server should detect an overflow condition for the selected queue, modify the following attributes:
   - `Queue Length`: Make sure that the Queue Length attribute indicates the maximum possible length that the execute queue can reach. This value should be higher than the normal operating length of the queue. By default, the Queue Length is set to 65536 entries.
   - `Queue Length Threshold Percent`: Enter the percentage (from 1–99) of the Queue Length size that can be reached before the server indicates an overflow condition for the queue. All actual queue length sizes below the threshold percentage are considered normal; sizes above the threshold percentage indicate an overflow. By default, WebLogic Server sets `Queue Length Threshold Percent` to 90 percent.
10. To specify how this server should address an overflow condition for the selected queue, modify the following attribute:
    - `Threads Increase`: Enter the number of threads WebLogic Server should add to this execute queue when it detects an overflow condition. If you specify zero threads (the default), the server changes its health state to "warning" in response to an overflow condition in the thread, but it does not allocate additional threads to reduce the workload.
11. To fine-tune the variable thread count of this execute queue, modify the following attributes:
    - `Threads Minimum`: Specify the minimum number of threads that WebLogic Server should maintain in this execute queue to prevent unnecessary overflow conditions. By default, WebLogic Server sets `Threads Minimum` to 5.
    - `Threads Maximum`: Specify the maximum number of threads that this execute queue can have; this value prevents WebLogic Server from creating an overly high thread count in the queue in response to continual overflow conditions. By default, WebLogic Server sets `Threads Maximum` to 400.
12. Click Apply to apply your changes.
13. Reboot the selected server to enable the new execute queue settings.

Detecting “Stuck” Threads

WebLogic Server automatically detects when a thread in an execute queue becomes “stuck.” Because a stuck thread cannot complete its current work or accept new work, the server logs a message each time it diagnoses a stuck thread. If all threads in an execute queue become stuck, the server changes its health state to either “warning” or “critical” depending on the execute queue:

- If all threads in the default queue become stuck, the server changes its health state to “critical.” (You can set up the Node Manager application to automatically shut down and restart servers in the critical health state. See “Node Manager Capabilities” in *Configuring and Managing WebLogic Server* for more information.)
- If all threads in `__weblogic_admin_html_queue`, `__weblogic_admin_rmi_queue`, or a user-defined execute queue become stuck, the server changes its health state to “warning.”

WebLogic Server diagnoses a thread as stuck if it is continually working (not idle) for a set period of time. You can tune a server’s thread detection behavior by changing the length of time before a thread is diagnosed as stuck, and by changing the frequency with which the server checks for stuck threads.

Note: Although you can change the criteria WebLogic Server uses to determine whether a thread is stuck, you cannot change the default behavior of setting the “warning” and “critical” health states when all threads in a particular execute queue become stuck.

To configure WebLogic Server thread detection behavior:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server instance whose thread detection behavior that you want to configure. Note that you configure stuck thread detection parameters on a per-server basis, rather than on a per-execute queue basis.
5. Select the Configuration →Tuning tab in the right pane.

6. Modify the following attributes as necessary to tune thread detection behavior for the server:
   - `Stuck Thread Max Time`: Enter the length of time, in seconds, that a thread must be continually working before this server diagnoses the thread as being stuck. By default, WebLogic Server considers a thread to be "stuck" after 600 seconds of continuous use.
   - `Stuck Thread Timer Interval`: Enter the length of time, in seconds, after which WebLogic Server periodically scans threads to see if they have been continually working for the length of time specified by `Stuck Thread Max Time`. By default, WebLogic Server sets this interval to 600 seconds.
7. Click Apply to apply your changes.
8. Reboot the server to use the new settings.

Tuning Connection Backlog Buffering

Use the `AcceptBacklog` attribute of the `Server` element in the `config.xml` file to set the number of connection requests the WebLogic Server instance will accept before refusing additional requests. The `AcceptBacklog` attribute specifies how many Transmission Control Protocol (TCP) connections can be buffered in a wait queue. This fixed-size queue is populated with requests for connections that the TCP stack has received, but the application has not accepted yet. The default value is 50 and the maximum value is operating system dependent.

To tune the Accept Backlog value from the Administration Console.

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server instance that you want to configure.
5. Select the Connection →Tuning tab.
6. Modify the default Accept Backlog value as necessary to tune how many TCP connections can be buffered in a wait queue:

- During operations, if many connections are dropped or refused at the client, and no other error messages are on the server, the Accept Backlog value might be set too low.
- If you are getting "connection refused" messages when you try to access WebLogic Server, raise the Accept Backlog value from the default by 25 percent. Continue increasing the value by 25 percent until the messages cease to appear.

7. Apply the changes.

How JDBC Connection Pools Enhance Performance

Establishing a JDBC connection with a DBMS can be very slow. If your application requires database connections that are repeatedly opened and closed, this can become a significant performance issue. WebLogic connection pools offer an efficient solution to the problem.

When WebLogic Server starts, connections from the connection pools are opened and are available to all clients. When a client closes a connection from a connection pool, the connection is returned to the pool and becomes available for other clients; the connection itself is not closed. There is little cost to opening and closing pool connections.

How many connections should you create in the pool? A connection pool can grow and shrink according to configured parameters, between a minimum and a maximum number of connections. The best performance occurs when the connection pool has as many connections as there are concurrent client sessions.

In addition to the following subsections, see "Performance Tuning Your JDBC Application" in *Programming WebLogic JDBC*.

Tuning JDBC Connection Pool Initial Capacity

The `InitialCapacity` attribute of the `JDBCConnectionPool` element enables you to set the number of physical database connections to create when configuring the pool. If the server cannot create this number of connections, the creation of this connection pool will fail.

During development, it is convenient to set the value of the `InitialCapacity` attribute to a low number. This helps the server start up faster.

In production systems, consider setting `InitialCapacity` equal to the `MaxCapacity` so that all database connections are acquired during server start-up. If `InitialCapacity` is less than `MaxCapacity`, the server needs to create additional database connections when its load is increased. When the server is under load, all resources should be working to complete requests as fast as possible, rather than creating new database connections.

Tuning JDBC Connection Pool Maximum Capacity

The `MaxCapacity` attribute of the `JDBCConnectionPool` element allows you to set the maximum number of physical database connections that a connection pool can contain. Different JDBC drivers and database servers might limit the number of possible physical connections.

In production, it is advisable that the number of connections in the pool equal the number of concurrent client sessions that require JDBC connections. The pool capacity is independent of the number of execute threads in the server. There may be many more ongoing user sessions than there are execute threads.

Caching Prepared and Callable Statements

For each connection pool that you create on a WebLogic Server, you can specify a statement cache size. When you set the statement cache size, WebLogic Server stores each statement used in applications and EJBs until it reaches the number of statements that you specify. For example, if you set the statement cache size to 10, WebLogic Server will store the first 10 prepared or callable statements called by applications or EJBs.

Using the statement cache can dramatically increase performance, but you must consider its limitations before you decide to use it. For more details, see Increasing Performance with the Prepared Statement Cache in the *Administration Console Online Help*.

Setting Java Parameters for Starting WebLogic Server

Java parameters must be specified whenever you start WebLogic Server. For simple invocations, this can be done from the command line with the `weblogic.Server` command. However, because the arguments needed to start a WebLogic Server from the command line can be lengthy and prone to error, we recommend that you incorporate the command into a script. To simply this process, you can modify the default values in the sample scripts that are provided with the WebLogic distribution to start WebLogic Server, as described in "Starting an Administration Server Using a Script".

The scripts for starting the Administration Server are called `startWLS.sh` (UNIX) and `startWLS.cmd` (Windows). These scripts are located in the `*WL_HOME*\server\bin` directory, where `*WL_HOME*` is the location in which you installed WebLogic Server.

You need to modify some default Java values in these scripts to fit your environment and applications. The important performance tuning parameters in these files are the `JAVA_HOME` parameter and the Java heap size parameters:

- Change the value of the variable `JAVA_HOME` to the location of your `JDK`. For example:

  ```
  set JAVA_HOME=C:\bea\jdk131_03
  ```

- For higher performance throughput, set the minimum java heap size equal to the maximum heap size. For example:

  ```
  "%JAVA_HOME%\bin\java" -hotspot –Xms512m –Xmx512m -classpath
  %CLASSPATH% -
  ```

  See "Specifying Heap Size Values" on page 2-7 for details about setting heap size options.

Setting Your Java Compiler

The standard Java compiler for compiling JSP servlets is `javac`. You can improve performance significantly by setting your server's java compiler to `sj` or `jikes` instead of `javac`. The following sections discuss this procedure and other compiler considerations.

Changing Compilers in the Administration Console

To change your compiler in the Administration Console:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server instance that you want to configure.
5. Select the Configuration →Compilers tab and enter the full path of the compiler in the Java Compiler field. For example:

   `c:\visualcafe31\bin\sj.exe`
6. Enter the full path to the JRE `rt.jar` library in the Append to the Classpath field. For example:

   `BEA_HOME``\jdk131_03\jre\lib\rt.jar`
7. Click Apply.
8. Restart your server for the new Java Compiler and Append to Classpath values to take effect.

Setting Your Compiler in weblogic.xml

In the `weblogic.xml` file, the `jsp-descriptor` element defines parameter names and values for servlet JSPs.

- Use the `compileCommand` parameter to specify the Java compiler for compiling the generated JSP servlets.
- Use the `precompile` parameter to configure WebLogic Server to precompile your JSPs when WebLogic Server starts up.

For more information about setting your server's java compiler in the `weblogic.xml` file, see the `jsp-descriptor` element.

Compiling EJB Container Classes

Use the `weblogic.appc` utility to compile EJB 2.0 and 1.1 container classes. If you compile `.jar` files for deployment into the EJB container, you must use `weblogic.appc` to generate the container classes. By default, ejbc uses the `javac` compiler. For faster performance, specify a different compiler (such as Symantec `sj`) using the `-compiler` flag.

For more information, see "WebLogic Server EJB Tools".

Compiling on UNIX

If you receive the following error message received when compiling JSP files on a UNIX machine:

```
failed: java.io.IOException: Not enough space
```

Try any or all of the following solutions:

- Add more RAM if you have only 256 MB.
- Raise the file descriptor limit, for example:

  ```
  set rlim_fd_max = 4096
  set rlim_fd_cur = 1024
  ```

- Use the `-native` flag to use native threads when starting the JVM.

Using WebLogic Server Clusters

A WebLogic Server cluster is a group of WebLogic Servers instances that together provide fail-over and replicated services to support scalable high-availability operations for clients. A cluster appears to its clients as a single server but is in fact a group of servers acting as one.

Scalability and High Availability

Scalability is the ability of a system to grow in one or more dimensions as more resources are added to the system. Typically, these dimensions include (among other things), the number of concurrent users that can be supported and the number of transactions that can be processed in a given unit of time.

Given a well-designed application, it is entirely possible to increase performance by simply adding more resources. To increase the load handling capabilities of WebLogic Server, add another WebLogic Server instance to your cluster—without changing your application. Clusters provide two key benefits that are not provided by a single server: scalability and availability.

WebLogic Server clusters bring scalability and high-availability to J2EE applications in a way that is transparent to application developers. Scalability expands the capacity of the middle tier beyond that of a single WebLogic Server or a single computer. The only limitation on cluster membership is that all WebLogic Servers must be able to communicate by IP multicast. New WebLogic Servers can be added to a cluster dynamically to increase capacity.

A WebLogic Server cluster guarantees high-availability by using the redundancy of multiple servers to insulate clients from failures. The same service can be provided on multiple servers in a cluster. If one server fails, another can take over. The ability to have a functioning server take over from a failed server increases the availability of the application to clients.

For complete information about clusters, see "Using WebLogic Server Clusters".

Caution: Provided that you have resolved all application and environment bottleneck issues, adding additional servers to a cluster should provide linear scalability. When doing benchmark or initial configuration test runs, isolate issues in a single server environment before moving to a clustered environment.

Performance Considerations for Multi-CPU Machines

With multi-processor machines, additional consideration must be given to the ratio of clustered WebLogic Server instances to the number of available CPUs. Because WebLogic Server has no built-in limit to the number of server instances that reside in a cluster, large, multi-processor servers, such as Sun Microsystems' Sun Enterprise 10000, can potentially host very large clusters or multiple clusters.

Before determining the optimal ratio of servers to CPUs, thoroughly test your application to determine:

- *Network Requirements*—If you discover that a Web application is primarily network I/O-bound, consider measures to increase network throughput before increasing the number of available CPUs. For truly network I/O-bound applications, installing a faster network interface card (NIC) may increase performance more than additional CPUs, because most CPUs would remain idle while waiting to read available sockets.

- *Disk I/O Requirements*—If you discover that a Web application is primarily disk I/O-bound, consider upgrading the number of disk spindles or individual disks and controllers before allocating additional CPUs.

In summary, ensure that a Web application is truly CPU-bound, rather than network or disk I/O-bound, before allocating additional CPUs.

For CPU-bound applications, begin performance tests using a ratio of one WebLogic Server instance for every CPU. If CPU utilization is consistently at or near 100 percent, increase the ratio of CPUs to servers (for example, allocate one WebLogic Server instance for ever two CPUs). For production systems, keep in mind that some spare CPU cycles should always be available to perform administration tasks.

Although the processing needs of Web applications varies, BEA has found that optimal results are generally obtained using a ratio of one Weblogic Server instance for every two CPUs.

Monitoring a WebLogic Server Domain

The tool for monitoring the health and performance of your WebLogic Server domain is the Administration Console. Using the Administration Console, you can view status and statistics for WebLogic Server resources such as servers, HTTP, the JTA subsystem, JNDI, security, CORBA connection pools, EJB, JDBC, and JMS.

For details, see "Monitoring a WebLogic Server Domain".

CHAPTER

4 Tuning WebLogic Server EJBs

The following sections describe how to tune WebLogic Server EJBs to match your application needs:

- "Setting Performance-Related weblogic-ejb-jar.xml Parameters" on page 4-1
- "Tuning In Response to Monitoring Statistics" on page 4-6
- "Other Performance Improvement Strategies" on page 4-10

Setting Performance-Related weblogic-ejb-jar.xml Parameters

The `weblogic-ejb-jar.xml` deployment file contains the WebLogic Server-specific EJB DTD that defines the concurrency, caching, clustering, and behavior of EJBs. It also contains descriptors that map available WebLogic Server resources to EJBs. WebLogic Server resources include security role names and data sources such as JDBC pools, JMS connection factories, and other deployed EJBs.

For information on how to modify the `weblogic-ejb-jar.xml` deployment file, see "Specifying and Editing the EJB Deployment Descriptors" in *Programming WebLogic Enterprise JavaBeans*.

Table 4-1 lists the `weblogic-ejb-jar.xml` file parameters that affect performance.

Table 4-1 Performance-Related weblogic-ejb-jar.xml Parameters

Element	For more information
`max-beans-in-free-pool`	See "Setting EJB Pool Size" on page 4-2.
`initial-beans-in-free-pool`	See "Tuning Initial Beans in Free Pool" on page 4-4.
`max-beans-in-cache`	See "Setting EJB Caching Size" on page 4-4.
`relationship-caching`	See "Relationship Caching Support" on page 4-5.
`concurrency-strategy`	See "Deferring Database Locking" on page 4-5.
`isolation-level`	See "Setting Transaction Isolation Level" on page 4-6.

The following sections describe these elements.

Setting EJB Pool Size

WebLogic Server maintains a free pool of EJBs for every stateless session bean class. The `max-beans-in-free-pool` element of the `weblogic-ejb-jar.xml` file defines the size of this pool. By default, `max-beans-in-free-pool` has no limit; the maximum number of beans in the free pool is limited only by the available memory.

This section discusses the following topics:

- "Allocating Pool Size for Session and Message Beans" on page 4-3
- "Allocating Pool Size for Entity Beans" on page 4-3
- "Tuning the Pool Size" on page 4-3

See also:

- max-beans-in-free-pool in *Programming WebLogic Enterprise JavaBeans*
- "Using max-beans-in-free-pool" in *Programming WebLogic Enterprise JavaBeans*

Allocating Pool Size for Session and Message Beans

When EJBs are created, the session bean instance is created and given an identity. When the client removes a bean, the bean instance is placed in the free pool. When you create a subsequent bean, you can avoid object allocation by reusing the previous instance that is in the free pool. The `max-beans-in-free-pool` element can improve performance if EJBs are frequently created and removed.

The EJB container creates new instances of message beans as needed for concurrent message processing. The `max-beans-in-pool` element puts an absolute limit on how many of these instances will be created. The container may override this setting according to the runtime resources that are available.

For the best performance for stateless session and message beans, use the default setting `max-beans-in-free-pool` element. The default allows you to run beans in parallel, using as many threads as possible. The only reason to change the setting is to limit the number of beans running in parallel.

Allocating Pool Size for Entity Beans

There is a pool of anonymous entity beans (i.e., beans without a primary key assigned to them) that is used to invoke finders and home methods, and to create entity beans. The `max-beans-in-free-pool` element also controls the size of this pool.

If you are running lots of finders or home methods or creating lots of beans, you may want to tune the `max-beans-in-free-pool` element so that there are enough beans available for use in the pool.

Tuning the Pool Size

Do not change the value of the `max-beans-in-free-pool` parameter unless you frequently create session beans, do a quick operation, and then throw them away. If you do this, enlarge your free pool by 25 to 50 percent and see if performance improves. If object creation represents a small fraction of your workload, increasing this parameter will not significantly improve performance. For applications where EJBs are database intensive, do not change the value of this parameter.

Caution: Tuning this parameter too high uses extra memory. Tuning it too low causes unnecessary object creation. If you are in doubt about changing this parameter, leave it unchanged.

Tuning Initial Beans in Free Pool

Use the `initial-beans-in-free-pool` element of the `weblogic-ejb-jar.xml` file to specify the number of stateless session bean instances in the free pool at startup.

If you specify a value for `initial-beans-in-free-pool`, WebLogic Server populates the free pool with the specified number of bean instances at startup. Populating the free pool in this way improves initial response time for the EJB, because initial requests for the bean can be satisfied without generating a new instance.

`initial-beans-in-free-pool` defaults to 0 if the element is not defined.

The `initial-beans-in-free-pool` element is described in *Programming WebLogic Enterprise JavaBeans*.

Setting EJB Caching Size

WebLogic Server enables you to configure the number of active beans that are present in the EJB cache (the in-memory space where beans exist).

The `max-beans-in-cache` element of the `weblogic-ejb-jar.xml` file specifies the maximum number of objects of this class that are allowed in memory. When `max-beans-in-cache` is reached, WebLogic Server passivates some EJBs that have not been recently used by a client. The `max-beans-in-cache` element also affects when EJBs are removed from the WebLogic Server cache.

Using this element sets the cache size for stateful session and entity beans similarly.

For more information, see “EJB Concurrency Strategy” in *Programming WebLogic Enterprise JavaBeans*

The `max-beans-in-cache` element is described in *Programming WebLogic Enterprise JavaBeans*.

Activation and Passivation of Stateful Session EJBs

Set the appropriate cache size with the `max-beans-in-cache` element to avoid excessive passivation and activation. Activation is the transfer of an EJB instance from secondary storage to memory. Passivation is the transfer of an EJB instance from memory to secondary storage. Tuning `max-beans-in-cache` too high consumes memory unnecessarily.

The EJB container performs passivation when it invokes the `ejbPassivate()` method. When the EJB session object is needed again, it is recalled with the `ejbActivate()` method. When the `ejbPassivate()` call is made, the EJB object is serialized using the Java serialization API or other similar methods and stored in secondary memory (disk). The `ejbActivate()` method causes the opposite.

The container automatically manages this working set of session objects in the EJB cache without the client's or server's direct intervention. Specific callback methods in each EJB describe how to passivate (store in cache) or activate (retrieve from cache) these objects. Excessive activation and passivation nullifies the performance benefits of caching the working set of session objects in the EJB cache—especially when the application has to handle a large number of session objects.

Relationship Caching Support

Relationship caching improves the performance of entity beans by loading related beans into the cache and avoiding multiple queries by issuing a join query for the related bean.

For more information on relationship caching, see Relationship Caching with Entity Beans.

Deferring Database Locking

WebLogic Server supports database locking and exclusive locking mechanisms. The default and *recommended* mechanism for EJB 1.1 and EJB 2.0 is database locking.

Database locking improves concurrent access to entity EJBs. The WebLogic Server container improves concurrent access by deferring locking services to the underlying database. Unlike exclusive locking, with deferred database locking, the underlying data store can provide finer granularity for locking EJB data, in most cases, and provide deadlock detection.

For details about database locking, see Database Concurrency Strategy in *Programming WebLogic Enterprise JavaBeans*.

You specify the locking mechanism used for an EJB by setting the `concurrency-strategy` deployment parameter in the `weblogic-ejb-jar.xml` file.

Setting Transaction Isolation Level

Data accessibility is controlled through the transaction isolation level mechanism. Transaction isolation level determines the degree to which multiple interleaved transactions are prevented from interfering with each other in a multi-user database system. Transaction isolation is achieved through use of locking protocols that guide the reading and writing of transaction data. This transaction data is written to the disk in a process called “serialization.” Lower isolation levels give you better database concurrency at the cost of less transaction isolation.

For more information, see the description of the `isolation-level` element of the `weblogic-ejb-jar.xml` file in *Programming WebLogic Enterprise JavaBeans*.

Refer to your database documentation for more information on the implications and support for different isolation levels.

Tuning In Response to Monitoring Statistics

The WebLogic Server Administration Console reports a wide variety of EJB runtime monitoring statistics, many of which are useful for tuning your EJBs. This section discusses how some of these statistics can help you tune the performance of EJBs.

To display the statistics in the Administration Console, see the following Console Help sections:

- Monitoring Stateless Session EJBs
- Monitoring Stateful Session EJBs
- Monitoring Entity EJBs
- Monitoring Message-Driven EJBs

Cache Miss Ratio

A high cache miss ratio could be indicative of an improperly sized cache. If your application uses a certain subset of beans (read primary keys) more frequently than others, it would be ideal to size your cache large enough so that the commonly used beans can remain in the cache as less commonly used beans are cycled in and out upon demand. If this is the nature of your application, you may be able to decrease your cache miss ratio significantly by increasing the maximum size of your cache.

If your application doesn't necessarily use a subset of beans more frequently than others, increasing your maximum cache size may not affect your cache miss ratio. We recommend testing your application with different maximum cache sizes to determine which give the lowest cache miss ratio. It is also important to keep in mind that your server has a finite amount of memory and therefore there is always a trade-off to increasing your cache size.

Lock Waiter Ratio

A high lock waiter ratio can indicate a sub optimal concurrency strategy for the bean. If acceptable for your application, a concurrency strategy of Database or Optimistic will allow for more parallelism than an Exclusive strategy and remove the need for locking at the EJB container level.

Since locks are generally held for the duration of a transaction, reducing the amount of time your transactions take will free up beans more quickly and may help reduce your lock waiter ratio.

Lock Timeout Ratio

The lock timeout ratio is closely related to the lock waiter ratio. If you are concerned about the lock timeout ratio for your bean, first take a look at the lock waiter ratio and our recommendations for reducing it (including possibly changing your concurrency strategy). If you can reduce or eliminate the number of times a thread has to wait for a lock on a bean, you will also reduce or eliminate the amount of timeouts that occur while waiting.

A high lock timeout ratio may also be indicative of an improper transaction timeout value. The maximum amount of time a thread will wait for a lock is equal to the current transaction timeout value.

If the transaction timeout value is set too low, threads may not be waiting long enough to obtain access to a bean and timing out prematurely. If this is the case, increasing the trans-timeout-seconds value for the bean may help reduce the lock timeout ratio.

Take care when increasing the trans-timeout-seconds, however, because doing so can cause threads to wait longer for a bean and threads are a valuable server resource. Also, doing so may increase the request time, as a request ma wait longer before timing out.

Pool Miss Ratio

If your pool miss ratio is high, you must determine what is happening to your bean instances. There are three things that can happen to your beans.

- They are in use.
- They were destroyed.
- They were removed.

Follow these steps to diagnose the problem:

1. Check your destroyed bean ratio to verify that bean instances are not being destroyed.

   Investigate the cause and try to remedy the situation.

2. Examine the demand for the EJB, perhaps over a period of time.

One way to check this is via the Beans in Use Current Count and Idle Beans Count. If demand for your EJB spikes during a certain period of time, you may see a lot of pool misses as your pool is emptied and unable to fill additional requests.

As the demand for the EJB drops and beans are returned to the pool, many of the beans created to satisfy requests may be unable to fit in the pool and are therefore removed. If this is the case, you may be able to reduce the number of pool misses by increasing the maximum size of your free pool. This may allow beans that were created to satisfy demand during peak periods to remain in the pool so they can be used again when demand once again increases.

Destroyed Bean Ratio

To reduce the number of destroyed beans, BEA recommends against throwing non-application exceptions from your bean code except in cases where you want the bean instance to be destroyed. A non-application exception is an exception that is either a java.rmi.RemoteException (including exceptions that inherit from RemoteException) or is not defined in the throws clause of a method of an EJB's home or component interface.

In general, you should investigate which exceptions are causing your beans to be destroyed as they may be hurting performance and be indicative of a problem with the EJB or a resource used by the EJB.

Pool Timeout Ratio

A high pool timeout ratio could be indicative of an improperly sized free pool. Increasing the maximum size of your free pool via the max-beans-in-free-pool setting will increase the number of bean instances available to service requests and may reduce your pool timeout ratio.

Another factor affecting the number of pool timeouts is the configured transaction timeout for your bean. The maximum amount of time a thread will wait for a bean from the pool is equal to the default transaction timeout for the bean. Increasing the `trans-timeout-seconds` setting in your `weblogic-ejb-jar.xml` file will give threads more time to wait for a bean instance to become available.

Users should exercise caution when increasing this value, however, since doing so may cause threads to wait longer for a bean and threads are a valuable server resource. Also, request time might increase because a request will wait longer before timing out.

Transaction Rollback Ratio

Begin investigating a high transaction rollback ratio by examining the Transaction Timeout Ratio. If the transaction timeout ratio is higher than you expect, try to address the timeout problem first.

An unexpectedly high transaction rollback ratio could be caused by a number of things. We recommend investigating the cause of transaction rollbacks to find potential problems with your application or a resource used by your application.

Transaction Timeout Ratio

A high transaction timeout ratio could be caused by the wrong transaction timeout value. For example, if your transaction timeout is set too low, you may be timing out transactions before the thread is able to complete the necessary work. Increasing your transaction timeout value may reduce the number of transaction timeouts.

You should exercise caution when increasing this value, however, since doing so can cause threads to wait longer for a resource before timing out. Also, request time might increase because a request will wait longer before timing out.

A high transaction timeout ratio could be caused by a number of things such as a bottleneck for a server resource. We recommend tracing through your transactions to investigate what is causing the timeouts so the problem can be addressed.

Other Performance Improvement Strategies

- "Combined Caching Support" on page 4-11
- "Batch Operations" on page 4-11

- "Distributing Transactions Across EJBs in a WebLogic Server Cluster" on page 4-11
- "Stateless Session EJB Life Cycle" on page 4-12
- "Stateful Session EJB Life Cycle" on page 4-13

Combined Caching Support

Combined caching support allows you to configure a single cache for use with multiple entity beans. This will help solve usability and performance problems. Previously, you were required to configure a separate cache for each entity bean that was part of an application. For more information on combined caching, see Combined Caching with Entity Beans.

Batch Operations

Batch inserts, updates and deletes improve the performance of container-managed persistence (CMP) bean creation by enabling the EJB container to perform multiple database operations for CMP beans in one SQL statement, thereby reducing network roundtrips. For more information on batch operations, see Batch Operations.

Distributing Transactions Across EJBs in a WebLogic Server Cluster

WebLogic Server provides additional transaction performance benefits for EJBs that reside in a WebLogic Server cluster. When a single transaction uses multiple EJBs, WebLogic Server attempts to use EJB instances from a single WebLogic Server instance, rather than using EJBs from different servers. This approach minimizes network traffic for the transaction.

In some cases, a transaction can use EJBs that reside on multiple WebLogic Server instances in a cluster. This can occur in heterogeneous clusters, where all EJBs have not been deployed to all WebLogic Server instances. In these cases, WebLogic Server uses a multitier connection to access the datastore, rather than multiple direct connections. This approach uses fewer resources, and yields better performance for the transaction.

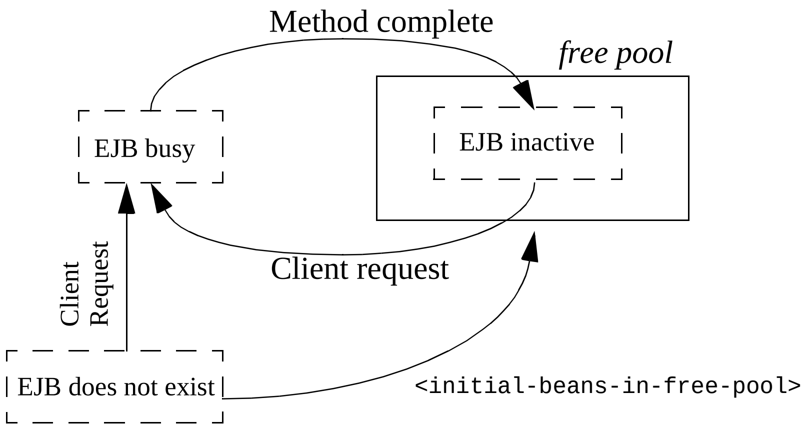
However, for best performance, the cluster should be homogeneous — all EJBs should reside on all available WebLogic Server instances.

Stateless Session EJB Life Cycle

WebLogic Server uses a *free pool* to improve performance and throughput for stateless session EJBs. The free pool stores *unbound* stateless session EJBs. Unbound EJB instances are instances of a stateless session EJB class that are not processing a method call.

The following figure illustrates the WebLogic Server free pool, and the processes by which stateless EJBs enter and leave the pool. Dotted lines indicate the “state” of the EJB from the perspective of WebLogic Server.

Figure 4-1 WebLogic Server free pool showing stateless session EJB life cycle



Stateful Session EJB Life Cycle

WebLogic Server uses a cache of bean instances to improve the performance of stateful session EJBs. The cache stores active EJB instances in memory so that they are immediately available for client requests. Active EJBs consist of instances that are currently in use by a client, as well as instances that were recently in use, as described in the following sections. The cache is unlike the free pool insofar as stateful session beans in the cache are bound to a particular client, while the stateless session beans in the free pool have no client association.

Passivating Stateful Session EJBs

To achieve high performance, WebLogic Server reserves the cache for EJBs that clients are currently using and EJBs that were recently in use. When EJBs no longer meet these criteria, they become eligible for *passivation*.

CHAPTER

5 Tuning WebLogic Server Applications

WebLogic Server only performs as well as the applications running on it. It is important to determine the bottlenecks that impede performance, as described in the following sections:

- "Using Performance Analysis Tools" on page 5-1
- "JDBC Application Tuning" on page 5-2
- "Managing Sessions" on page 5-3
- "Using Execute Queues to Control Thread Usage" on page 5-4

Using Performance Analysis Tools

This section is a quick reference for using the OptimizeIt™ and JProbe™ profilers with WebLogic Server.

A profiler is a performance analysis tool that allows you to reveal hot spots in the application that result in either high CPU utilization or high contention for shared resources. For a list of common profilers, see "Performance Analysis Tools" on page A-4.

Using the JProbe Profiler

The JProbe Profiler with Memory Debugger from Sitraka is a family of products that provide the capability to detect performance bottlenecks, perform code coverage and other metrics.

Sitraka provides the following Application Server Integration Portals for JProbe, which list the WebLogic Server releases certified to work with JProbe 3.0 and 4.0, and provide detailed instructions on how to integrate JProbe with WebLogic Server.

- JProbe 3.0.x Integration Portal for J2EE.
- JProbe 4.0 Integration Portal for J2EE.

Using the OptimizeIt Profiler

The OptimizeIt Profiler from Borland is a performance debugging tool for Solaris and Windows platforms.

Borland provides detailed J2EE Integration Tutorials for the supported versions of OptimizeIt Profiler that work with WebLogic Server.

JDBC Application Tuning

Most performance gains or losses in a database application are determined by how the application is designed. The number and location of clients, size and structure of DBMS tables and indexes, and the number and types of queries all affect application performance.

For more information on optimizing your applications for JDBC, see “Performance Tuning Your JDBC Application” in *Programming WebLogic JDBC*.

JDBC Optimization for Type-4 MS SQL Driver

When using the type-4 MS SQL driver, it may be much faster to create and execute an SQL statement either without parameters or with parameter values converted to their string counterparts and added as appropriate to the string, rather than declaring a long series of `setXXX()` calls, followed by `execute()`.

For more information, see "Configuring and Using WebLogic jDriver for Microsoft SQL Server"

Managing Sessions

Optimize your application so that it does as little work as possible when handling session persistence and sessions.

Managing Session Persistence

In-memory replication is up to 10 times faster than JDBC-based persistence for session state. Use in-memory replication, if possible.

If you are using JDBC-based persistence, optimize your code so that it has as high a granularity for session state persistence as possible. In the case of JDBC-based persistence, every session "put" operation that you use in your code results in a database write of the entire object.

Keep the number of "puts" that you use during your HTTP session to a minimum.To minimize how often information is persisted during a given session, examine your "puts" and, if possible, combine them into a single, large "put".

For more information, see:

- "Configuring Session Persistence" in *Assembling and Configuring Web Applications*
- "HTTP Session State Replication" in *Using WebLogic Sever Clusters*.

- "In-Memory Replication for Stateful Session EJBs" in *Programming WebLogic EJB.*
- "Using a Database for Persistent Storage (JDBC Persistence)" in *Assembling and Configuring Web Applications*,.

Minimizing Sessions

Configuring how WebLogic Server manages sessions is a key part of tuning your application for best performance. Consider the following:

- Use of sessions involves a scalability trade-off.
- Use sessions sparingly.

  Use sessions only for state that cannot realistically be kept on the client or if URL rewriting support is required. Keep simple bits of state, such as a user's name, directly in cookies. You might also write a wrapper class to "get" and "set" these cookies, in order to simplify the work of servlet developers working on the same project.
- Keep frequently used values in local variables.
- Put aggregate objects rather than multiple single objects into the session where possible.

See "Setting Up Session Management" in *Assembling and Configuring Web Applications*.

Using Execute Queues to Control Thread Usage

You can fine-tune an application's access to execute threads (and thereby optimize or throttle its performance) by using multiple execute queues in WebLogic Server. However, keep in mind that unused threads represent significant wasted resources in a Weblogic Server system. You may find that available threads in configured execute queues go unused, while applications in other queues sit idle waiting for threads to become available. In such a situation, the division of threads into multiple queues may yield poorer overall performance than having a single, default execute queue.

Default WebLogic Server installations are configured with a default execute queue, which is used by all applications running on the server instance. You may want to configure additional queues to:

- **Optimize the performance of critical applications.** For example, you can assign a single, mission-critical application to a particular execute queue, guaranteeing a fixed number of execute threads. During peak server loads, nonessential applications may compete for threads in the default execute queue, but the mission-critical application has access to the same number of threads at all times.
- **Throttle the performance of nonessential applications.** For an application that can potentially consume large amounts of memory, assigning it to a dedicated execute queue effectively limits the amount of memory it can consume. Although the application can potentially use all threads available in its assigned execute queue, it cannot affect thread usage in any other queue.
- **Remedy deadlocked thread usage.** With certain application designs, deadlocks can occur when all execute threads are currently utilized. For example, consider a servlet that reads messages from a designated JMS queue. If all execute threads in a server are used to process the servlet requests, then no threads are available to deliver messages from the JMS queue. A deadlock condition exists, and no work can progress. Assigning the servlet to a separate execute queue avoids potential deadlocks, because the servlet and JMS queue do not compete for thread resources.

Be sure to monitor each execute queue to ensure proper thread usage in the system as a whole. See “Setting Thread Count” on page 3-4 for general information about optimizing the number of threads.

Creating Execute Queues

An execute queue represents a named collection of execute threads that are available to one or more designated servlets, JSPs, EJBs, or RMI objects. An execute queue is represented in the domain `config.xml` file as part of the `Server` element. For example, an execute queue named `CriticalAppQueue` with four execute threads appears in the `config.xml` file as follows:

```
...
<Server
 Name="examplesServer"
 ListenPort="7001"
 NativeIOEnabled="true"/>
 <ExecuteQueue Name="default"
  ThreadCount="15"/>
 <ExecuteQueue Name="CriticalAppQueue"
  ThreadCount="4"/>
 ...
</Server>
```

To configure a new execute queue using the Administration Console:

1. Start the Administration Server if it is not already running.
2. Access the Administration Console for the domain.
3. Click the Servers node in the left pane to display the servers configured in your domain.
4. Click the name of the server instance where you will add the execute queue.
5. Select the Monitoring →General tab.
6. Click the Monitor All Active Queues text link to display the execute queues that the selected server uses.
7. Click the Configure Execute Queue text link to display the execute queues that you can modify.
8. Click the Configure a New Execute Queue link.
9. On the Execute Queue Configuration tab modify the following attributes or accept the system defaults:
   - `Queue Length`: Always leave the Queue Length at the default value of 65536 entries. The Queue Length specifies the maximum number of

simultaneous requests that the server can hold in the queue. The default of 65536 requests represents a very large number of requests; outstanding requests in the queue should rarely, if ever reach this maximum value.

If the maximum Queue Length is reached, WebLogic Server automatically doubles the size of the queue to account for the additional work. Note, however, that exceeding 65536 requests in the queue indicates a problem with the threads in the queue, rather than the length of the queue itself; check for stuck threads or an insufficient thread count in the execute queue.

- `Queue Length Threshold Percent`: Enter the percentage (from 1–99) of the Queue Length size that can be reached before the server indicates an overflow condition for the queue. All actual queue length sizes below the threshold percentage are considered normal; sizes above the threshold percentage indicate an overflow. When an overflow condition is reached, WebLogic Server logs an error message and increases the number of threads in the queue by the value of the Threads Increase attribute to help reduce the workload.

  By default, the Queue Length Threshold Percent value is 90 percent. In most situations, you should leave the value at or near 90 percent, to account for any potential condition where additional threads may be needed to handle an unexpected spike in work requests. Keep in mind that Queue Length Threshold Percent must not be used as an automatic tuning parameter—the threshold should never trigger an increase in thread count under normal operating conditions.

- `Thread Count`: Specify the number of threads assigned to this queue. If you do not need to use more than 15 threads (the default) for your work, do not change the value of this attribute. (For more information, see “Should You Modify the Default Thread Count?” on page 3-4.)

- `Threads Increase`: Enter the number of threads WebLogic Server should add to this execute queue when it detects an overflow condition. If you specify zero threads (the default), the server changes its health state to “warning” in response to an overflow condition in the thread, but it does not allocate additional threads to reduce the workload.

  Note that if WebLogic Server increases the number of threads in response to an overflow condition, the additional threads remain in the execute queue until the server is rebooted. In general, you should monitor the error log to determine the cause of overflow conditions, and reconfigure the thread count as necessary to prevent similar conditions in the future. Do not use the combination of Threads Increase and Queue Length Threshold Percent as an automatic tuning tool; doing so generally results in the execute queue allocating more threads than necessary and suffering from poor performance due to context switching.

- `Threads Minimum`: Specify the minimum number of threads that WebLogic Server should maintain in this execute queue to prevent unnecessary overflow conditions. By default, the Threads Minimum is set to 5.
- `Threads Maximum`: Specify the maximum number of threads that this execute queue can have; this value prevents WebLogic Server from creating an overly high thread count in the queue in response to continual overflow conditions. By default, the Threads Maximum is set to 400.
- `Thread Priority`: Specify the priority of the threads associated with this queue. By default, the Thread Priority is set to 5.

10. Click Create to create the new execute queue.

11. Reboot the server to use the new settings.

Assigning Servlets and JSPs to Execute Queues

You can assign a servlet or JSP to a configured execute queue by identifying the execute queue name in the initialization parameters. Initialization parameters appear within the `init-param` element of the servlet's or JSP's deployment descriptor file, `web.xml`. To assign an execute queue, enter the queue name as the value of the `wl-dispatch-policy` parameter, as in the example:

```
<servlet>
   <servlet-name>MainServlet</servlet-name>
   <jsp-file>/myapplication/critical.jsp</jsp-file>
   <init-param>
      <param-name>wl-dispatch-policy</param-name>
      <param-value>CriticalAppQueue</param-value>
   </init-param>
</servlet>
```

See "Initializing a Servlet" in *Programming WebLogic HTTP Servlets* for more information about specifying initialization parameters in `web.xml`.

Assigning EJBs and RMI Objects to Execute Queues

To assign an RMI object to a configured execute queue, use the `-dispatchPolicy` option to the `rmic` compiler. For example:

```
java weblogic.rmic -dispatchPolicy CriticalAppQueue ...
```

To assign an EJB object to a configured execute queue, use the `-dispatchPolicy` option with the `appc` utility. `appc` passes this option and its argument to `rmic` when compiling the EJB.

CHAPTER

A Related Reading: Performance Tools and Information

The following sections provide an extensive performance-related reading list:

BEA Systems, Inc. Information

- For general information about BEA Systems, see the BEA Web site

- BEA WebLogic Server Documentation page
- BEA WebLogic Platform Documentation page
- BEA's dev2dev Web site

- BEA WebLogic Server White Papers
- *J2EE Design Considerations for WebLogic Server*, BEA White Paper, 2000
- *J2EE Applications and BEA WebLogic Server* by Michael Girdley, Rob Woollen, Sandra Emerson, 2001
- *Professional J2EE Programming with BEA WebLogic Server* by Paco Gomez, Peter Zadrozny, 2000
- *BEA WebLogic Server Bible* by Joe Zuffoletto, et al, 2002
- J2EE Performance Testing with BEA WebLogic Server by Peter Zadrozny, Philip Aston, and Ted Osborne 2002

Sun Microsystems Information

- For general information about Sun Microsystems, see Sun's Web site
- Sun Microsystems Performance Information
- Java Standard Edition Platform Documentation
- Java 2 SDK, Standard Edition Documentation
- *Solaris Tunable Parameters Reference Manual*

- For BEA WebLogic Server and Solaris-specific details, see Sun Microsystems SPARC Solaris 2.5.1, 2.6, 2.7 and Sun Microsystems SPARC with Solaris 8 on the BEA Certifications Pages
- For more about Solaris configuration, check the Solaris FAQ

- *Sun Performance and Tuning Java and the Internet* by Adrian Cockcroft, et al, 1997
- *Solaris 7 Performance Administration Tools* by Frank Cervone, 2000

Hewlett-Packard Company Information

- General Hewlett-Packard information
- For BEA WebLogic Server and HP-UX-specific details, see Hewlett-Packard HP/9000 with HP-UX 11.0 and 11i on the BEA Certifications Pages
- Java Performance Tuning on HP-UX
- Hewlett Packard JMeter, a tool for analyzing profiling information
- GlancePlus system performance diagnostic tool
- HPjconfig Java system configuration tool

Microsoft Information

- General Microsoft information
- Windows 2000 Performance Tuning White Paper
- SQL-Server-Performance.Com, Microsoft SQL Server Performance Tuning and Optimization

- *Microsoft SQL Server 2000 Performance Optimization and Tuning Handbook*, by Ken England, 2001, Digital Press

Web Performance Tuning Information

- Apache Performance Notes
- iPlanet Web Server 4.0 Performance Tuning, Sizing, and Scaling
- *The Art and Science of Web Server Tuning with Internet Information Services 5.0*
- *Web Performance Tuning: Speeding Up the Web*, by Patrick Killelea, Linda Mui (Editor), O'Reilly Nutshell, 1998
- *Capacity Planning for Web Performance: Metrics, Models, and Methods*, by Daniel A. Menasce, Virgilio A. F. Almeida, Prentice Hall PTR, 1998

Network Performance Tools

- TracePlus/Ethernet, a network packet analysis tool for Windows 95/98/ME, NT 4.x, Windows 2000/XP

Performance Analysis Tools

A profiler is a performance analysis tool that allows you to reveal hot spots in the application that result in either high CPU utilization or high contention for shared resources. Some common profilers are:

- OptimizeIt Java Performance Profiler, a performance debugging tool for Solaris and Windows

- JProbe Profiler with Memory Debugger, a family of products that provide the capability to detect performance bottlenecks, perform code coverage and other metrics
- Product Review: OptimizeIt vs. JProbe, *Journal of Object Oriented Programming*, June 2001
- Hewlett Packard JMeter, a tool for analyzing profiling information
- Topaz, Mercury Interactive's application performance management solution
- SE Toolkit, a performance analysis tool kit

Benchmarking Information

- SPECjbb2000, a software benchmark product developed by the Standard Performance Evaluation Corporation (SPEC). SPECjbb2000 is designed to measure a system's ability to run Java server applications.
- ECPerf Benchmark Kit, a software benchmark product developed under the Java Community Process^{SM} Program that is designed to measure performance and scalability and assist the J2EE user in understanding J2EE scalability and tuning.
- SPECjAppServer2001 (Java Application Server), a client/server benchmark for measuring the performance of Java Enterprise Application Servers using a subset of J2EE API's in a complete end-to-end web application.

Java Virtual Machine (JVM) Information

- WebLogic JRockit for Windows and Linux
- JVM Corner at artima.com
- Hewlett-Packard Compulsory Tuning Steps, Garbage Collection and Heap Size

- Sun Microsystems FAQ about Java HotSpot technology and about performance in general
- Tuning Garbage Collection with the 1.3.1 Java Virtual Machine
- Java HotSpot VM Options, a Sun Microsystems document provides information on the command-line options and environment variables that can affect the performance characteristics of the HotSpot JVM.
- The Java HotSpot Client and Server Virtual Machines for J2SE 1.3
- Which Java VM scales best? From JavaWorld, results of a VolanoMark 2.0 server benchmark show how 12 virtual machines stack up.
- *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* by Richard Jones, Rafael D Lins, John Wiley & Sons, 1999

Enterprise JavaBeans Information

- *Programming WebLogic Enterprise JavaBeans*
- *Enterprise JavaBeans, Second Edition*, by Richard Monson-Haefel, Mike Loukides (Editor), 2000
- *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*, by Ed Roman, 1999
- TheServerSide.com, a free online community dedicated to Enterprise JavaBeans (EJBs) and J2EE.
- *Seven Rules for Optimizing Entity Beans*, by Akara Sucharitakul, Java Developer Connection, 2001

Java Message Service (JMS) Information

- *Programming WebLogic JMS*

- "Configuring JMS" in the WebLogic Server *Administration Console Online Help*
- "Tuning JMS" in the WebLogic Server *Administration Console Online Help*
- "Monitoring JMS" in the WebLogic Server *Administration Console Online Help*
- Sun Microsystems' JMS Specification

General Performance Information

- Jack Shirazi's Java Performance Tuning Web site
- The Software Testing and Quality Engineering Magazine, Web Application Scalability, "Avoiding Scalability Shock" by Bill Shea, May/June 2000
- High-Performance Java Platform Computing™ by Thomas W. Christopher, George K. Thiruvathukal, 2000
- *Performance and Idiom Guide* by Craig Larman and Rhett Guthrie, 1999

B Benchmark Tuning Examples for WebLogic Server

The following sections provide recommendations for improving the out-of-the-box performance of WebLogic Server when running the ECPerf or SPECjAppServer 2001/2002 benchmarks. Optimal WebLogic Server production tuning values vary according to your environment and applications.

- "Tuning an Intel Xeon System" on page B-1
- "Tuning a Sun UltraSparc III System" on page B-3

Tuning an Intel Xeon System

On a system using Intel's Xeon (Pentium 4) processor, the following tuning recommendations may provide up to a 700% performance improvement over WebLogic Server's default out-of-the-box tuning configuration. These recommendations are based on the following hardware and operating system configuration:

- 4-processor, 1.6 GHz Intel Xeon (Pentium 4)
- Hyper-threading technology enabled
- 4GB of memory

- Windows 2000 Advanced Server

JVM Tuning Tips

Use the BEA JRockit JVM instead of the Sun JVM that comes bundled with WebLogic Server. For more information on tuning JRockit, see the BEA JRockit Performance Tuning Guide.

- Use the following JRockit garbage collection and memory management options:
  - `-Xnativethreads` (JRockit native thread system)
  - `-Xgc:parallel` (parallel garbage collector)
  - `-Xallocationtype:local` (local thread allocation)
- Increase the minimum and maximum heap sizes to 1536MB (`-Xms=1536m -Xmx=1536m`).
- Specify the young generation (nursery) heap size as 512MB (`-Xns=512m`).

WebLogic Server Tuning Tips

Implement the following tuning recommendations on your WebLogic Server instance. For more information on tuning WebLogic Server parameters, see Chapter 3, “Tuning WebLogic Server.”

- Run *one* instance of WebLogic Server on the Intel Pentium machine.
- Increase the default `ExecuteQueue` parameter to 27 threads.
- Increase the JDBC connection pool `InitialCapacity` and `MaxCapacity` database connection parameters to 40.
- Increase the JDBC connection pool `PreparedStatementCacheSize` parameter to 300.

Tuning a Sun UltraSparc III System

On a system using Sun Microsystems' UltraSparc III processor, the following tuning recommendations may provide up to a 560% performance improvement over WebLogic Server's default out-of-box tuning configuration. These recommendations are based on the following hardware and operating system configuration:

- 4-processor, 900 MHz Sun UltraSparc III
- 4GB of memory
- Solaris 8

JVM Tuning Tips

The following tuning recommendations apply to the Sun Hotspot JVM that comes bundled with WebLogic Server. For more information on tuning the Hotspot JVM, see Chapter 2, "Tuning Java Virtual Machines (JVMs)."

- Use the HotSpot Server VM option (`-server`) instead of the HotSpot Client VM (`-client`).
- Increase the minimum and maximum heap sizes to 1536MB (`-Xms=1536m -Xmx=1536m`).
- Specify the New generation minimum and maximum heap sizes as 350MB (`-XX:NewSize=350m -XX:MaxNewSize=350m`).
- Specify the New generation survivor ratio as 10 (`-XX:SurvivorRatio=10`).

WebLogic Server Tuning Tips

Implement the following tuning recommendations on your WebLogic Server instances. For more information on tuning WebLogic Server parameters, see Chapter 3, "Tuning WebLogic Server."

- Run *two* instances of WebLogic Server on the UltraSparc III machine.

- Specify `-Dweblogic.PosixSocketReaders=1` on the command line when you start the WebLogic Server instances.
- Increase the JDBC connection pool `InitialCapacity` and `MaxCapacity` database connection parameters to 25.
- Increase the JDBC connection pool `PreparedStatementCacheSize` parameter to 300.

Index