Oracle9*i* Application Server

Oracle HTTP Server *powered by Apache* Performance Guide

Release 1.0.2.2 for Windows NT

May 2001

Part No. A86676-02

Oracle9*i* Application Server

Oracle HTTP Server *powered by Apache* Performance Guide, Release 1.0.2.2 for WIndows NT

Part No. A86676-02

Contributors: Sharon Malek, Carol Orange, Leela Rao

Contents

Send Us Your Comments

Oracle9*i* Application Server Oracle HTTP Server *powered by Apache* Performance Guide, Release 1.0.2.2 for WIndows NT

Part No. A86676-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail - iasdocs_us@oracle.com
- Fax - (650) 506-7409 Attn: Oracle9*i* Application Server Documentation Manager
- Postal service:

  Oracle Corporation
  Oracle9*i* Application Server Documentation Manager
  500 Oracle Parkway, M/S 2op4
  Redwood Shores, CA 94065 USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This guide discusses configuration and performance tuning of the Oracle HTTP Server *powered by Apache*.

There are many sources of information on configuring and tuning web servers, Apache in particular. This guide refers to those sources when expedient, and, where practical, quantifies the performance gains resulting from configuration actions found in those sources. Any recommendations not validated by Oracle in-house testing are cited as such, with attribution to the original source.

All in-house tests detailed in this guide were run on a dedicated 100 Mbps network, in order to achieve repeatable test results. Your results will vary based on network configuration and contention characteristics.

This preface contains these topics:

- Audience
- Organization
- Related Documentation
- Conventions

Audience

This guide is written for Oracle9*i* Application Server developers and system administrators who are responsible for configuring and tuning the Oracle HTTP Server *powered by Apache*.

To use this document, you need a working knowledge of web server administration and performance tuning concepts.

Organization

This document contains:

Chapter 1, "Performance Overview"

Describes performance and tuning concepts and terminology, with a description of the Oracle HTTP Server components in the Oracle9*i* Application Server architecture.

Chapter 2, "Monitoring Your Web Server"

Discusses the importance of monitoring to performance tuning, and describes tools and processes for gathering information about the web server and operating system software.

Chapter 3, "Sizing and Configuration"

Provides guidelines and approaches to sizing and configuration to meet performance goals.

Chapter 4, "Optimizing HTTP Server Performance"

Discusses tuning parameters to improve HTTP server performance and the effects of caching and logging on performance.

Chapter 5, "Optimizing Apache JServ"

Discusses performance and load balancing for Apache JServ, and optimizing the performance of OracleJSP pages.

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Application Server Overview Guide*
- *OracleJavaServer Pages Developer's Guide and Reference*

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://technet.oracle.com/membership/index.htm`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://technet.oracle.com/docs/index.htm`

The following sources provide additional information on topics found in this guide:

- *Windows NT Performance Tuning and Optimization* by Kenton Gardinier, Berkeley: Osborne / McGraw-Hill, 1998
- For information on the mod_status utility, see

  `http://www.oreillynet.com/pub/a/apache/2000/04/21/wrangler.html`
  `http://www.apache.org/docs/mod/mod_status.html`

- For information on the LogLevel directive, see

  `http://www.apache.org/docs/mod/core.html#loglevel`

- For information on Apache web server performance, see Dale Gaudet's *Apache Performance Notes* at

  `http://www.apache.org/docs/misc/perf-tuning.html`

- For information about performance and the HTTP/1.1 protocol, see

  `http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html`

- For more information on the FIN_WAIT_2 state, see

  `http://apache.put.poznan.pl/misc/fin_wait_2.html`

Conventions

This section describes the conventions used in the text and code examples of the this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	The C datatypes such as **ub4**, **sword**, or **OCINumber** are valid. When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles, emphasis, syntax clauses, or placeholders.	*Oracle8i Concepts* You can specify the *parallel_clause*. Run `U`*`old_release`*`.SQL` where *old_release* refers to the release you installed prior to upgrading.
lowercase monospace (fixed-width font)	Lowercase monospace typeface indicates executables and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, user names and roles, program units, and parameter values.	Enter `sqlplus` to open SQL*Plus. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (digits [ , precision ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		 `[COMPRESS | NOCOMPRESS]`

...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS subquery;`
	■ That you can repeat a portion of the code	`SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as it is shown.	`acctbal NUMBER(11,2);`
		 `acct     CONSTANT NUMBER(4) := 3;`

Italics	Italicized text indicates variables for which you must supply particular values.	CONNECT SYSTEM/*system_password*

Convention	Meaning	Example

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.	`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`

1 Performance Overview

This chapter discusses performance and tuning concepts, and briefly describes Oracle9*i* Application Server architecture.

This chapter contains the following sections:

Performance Terms

Following are performance terms used in this book:

concurrency	The ability to handle multiple requests simultaneously. Threads and processes are examples of concurrency mechanisms.
contention	Competition for resources.
hash	A number generated from a string of text with an algorithm. The hash value is substantially smaller than the text itself. Hash numbers are used for security and for faster access to data.
latency	The time that one system component spends waiting for another component in order to complete the entire task. Latency can be defined as wasted time. In networking contexts, latency is defined as the travel time of a packet from source to destination.
response time	The time between the submission of a request and the receipt of the response.
scalability	The ability of a system to provide **throughput** in proportion to, and limited only by, available hardware resources. A scalable system is one that can handle increasing numbers of requests without adversely affecting response time and **throughput**.
service time	The time between the receipt of a request and the completion of the response to the request.
think time	The time the user is not engaged in actual use of the processor.
throughput	The number of requests processed per unit of time.
wait time	The time between the submission of the request and initiation of the request.

What is Performance Tuning?

Performance must be built in. You must anticipate performance requirements during application analysis and design, and balance the costs and benefits of optimal performance. This section introduces some fundamental concepts:

- Response Time
- System Throughput
- Wait Time
- Critical Resources
- Effects of Excessive Demand
- Adjustments to Relieve Problems

See Also: "Setting Performance Targets" on page 1-8 for a discussion on performance requirements and determining what parts of the system to tune.

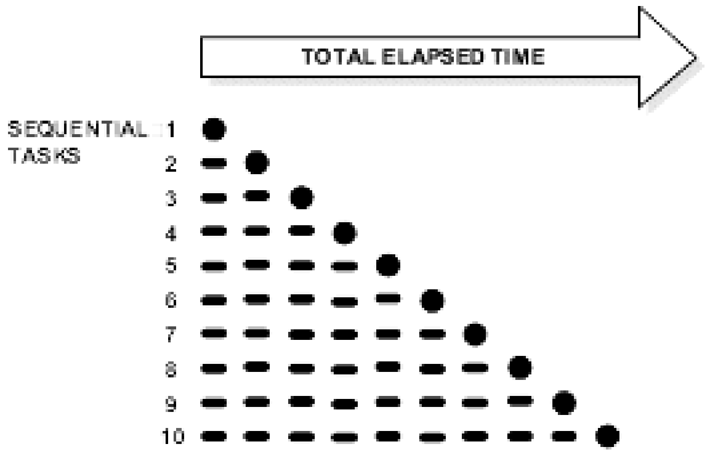
- Response Time

Because **response time** equals **service time** plus **wait time**, you can increase performance in this area by:

- Reducing **wait time**
- Reducing **service time**

Figure 1–1 illustrates ten independent tasks competing for a single resource.

Figure 1–1 Sequential processing of independent tasks



In this example, only task 1 runs without waiting. Task 2 must wait until task 1 has completed; task 3 must wait until tasks 1 and 2 have completed, and so on. (Although the figure shows the independent tasks as the same size, the size of the tasks will vary.)

In parallel processing with multiple resources, more resources are available to the tasks. Each independent task executes immediately using its own resource: no **wait time** is involved.

System Throughput

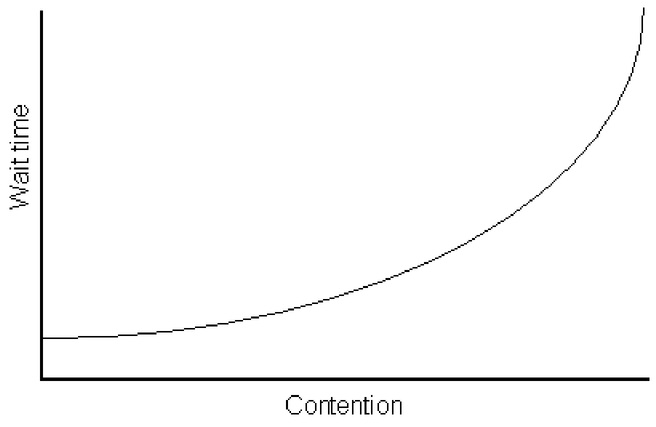
System **throughput** is the amount of work accomplished in a given amount of time. You can increase **throughput** by:

- Reducing **service time**
- Reducing overall **response time** by increasing the amount of scarce resources available. For example, if the system is CPU bound, and you can add more CPUs.

Wait Time

While the **service time** for a task may stay the same, **wait time** will lengthen with increased **contention**. If many users are waiting for a service that takes one second, the tenth user must wait 9 seconds. Figure 1–2 shows the relationship between **wait time** and resource **contention**.

Figure 1–2 Wait time rising with increased contention for a resource



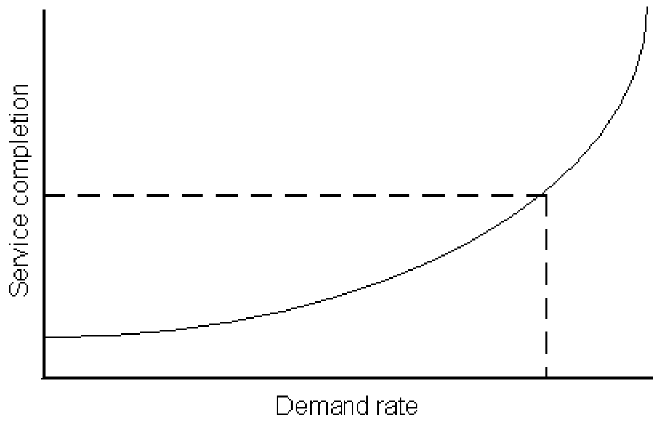
Critical Resources

Resources such as CPU, memory, I/O capacity, and network bandwidth are key to reducing **service time**. Adding resources increases **throughput** and reduces **response time**. Performance depends on these factors:

- How many resources are available?
- How many clients need the resource?
- How long must they wait for the resource?
- How long do they hold the resource?

Figure 1–3 shows that as the number of units requested rises, the time to service completion rises.

Figure 1–3 Time to service completion vs. demand rate



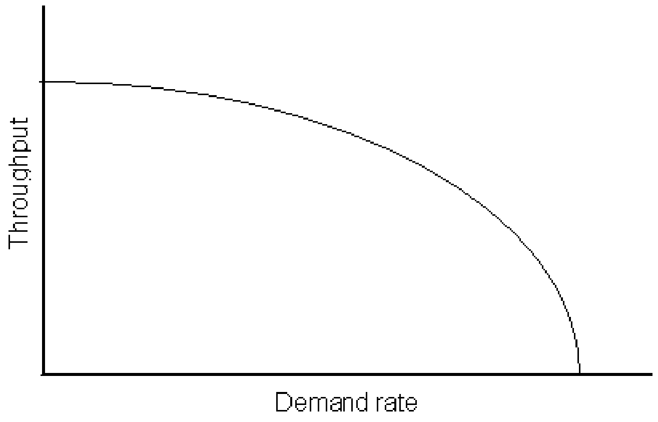
To manage this situation, you have two options:

- Limit demand rate to maintain acceptable **response time**s
- Add resources

Effects of Excessive Demand

Excessive demand increases **response time** and reduces **throughput**, as shown in Figure 1–4. If there is any possibility of the demand rate exceeding the achievable **throughput**, then determine which parameters should be adjusted (such as `ThreadsPerChild` in the Oracle HTTP Server and `security.maxConnections` in JServ) and change the configuration accordingly.

Figure 1–4 Increased Demand/Reduced Throughput



Adjustments to Relieve Problems

Performance problems can be relieved by making adjustments in the following areas:

unit consumption	Reducing the resource (CPU, memory) consumption of each request can improve performance. This might be achieved by pooling and caching.
functional demand	Rescheduling or redistributing the work will relieve some problems.
capacity	Increasing or reallocating resources (such as CPUs) relieves some problems.

Setting Performance Targets

Whether you are designing or maintaining a system, you should set specific performance goals so that you know how and what to optimize. If you alter parameters without a specific goal in mind, you can waste time tuning your system without significant gain.

An example of a specific performance goal is an order entry **response time** under three seconds. If the application does not meet that goal, identify the cause (for example, I/O **contention**), and take corrective action. During development, test the application to determine if it meets the designed performance goals.

Tuning usually involves a series of trade-offs. Once you have determined the bottlenecks, you may have to modify performance in some other areas to achieve the desired results. For example, if I/O is a problem, you may need to purchase more memory or more disks. If a purchase is not possible, you may have to limit the **concurrency** of the system to achieve the desired performance. However, if you have clearly defined goals for performance, the decision on what to trade for higher performance is simpler because you have identified the most important areas.

Setting User Expectations

Application developers, database administrators, and system administrators must be careful to set appropriate performance expectations for users. When the system carries out a particularly complicated operation, **response time** may be slower than when it is performing a simple operation. Users should be made aware of which operations might take longer.

Evaluating Performance

With clearly defined performance goals, you can readily determine when performance tuning has been successful. Success depends on the functional objectives you have established with the user community, your ability to measure whether or not the criteria are being met, and your ability to take corrective action to overcome any exceptions.

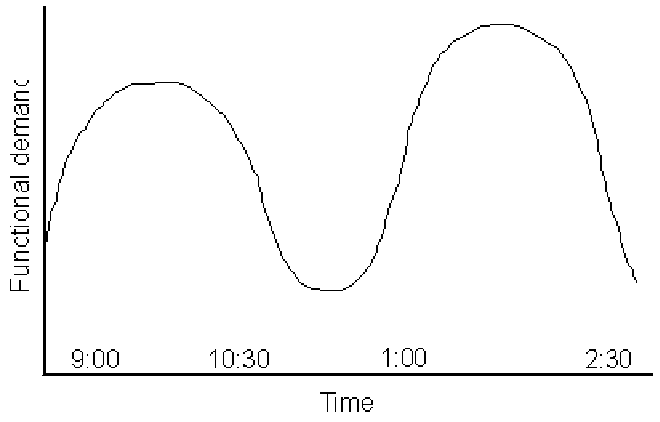
Ongoing performance monitoring enables you to maintain a well tuned system. Keeping a history of the application's performance over time enables you to make useful comparisons. With data about actual resource consumption for a range of loads, you can conduct objective **scalability** studies and from these predict the resource requirements for anticipated load volumes.

Performance Methodology

Achieving optimal effectiveness in your system requires planning, monitoring, and periodic adjustment. The first step in performance tuning is to determine the goals you need to achieve and to design effective usage of available technology into your applications. After implementing your system, it is necessary to periodically monitor and adjust your system For example, you might want to ensure that 90% of the users experience **response times** no greater than 5 seconds and the maximum **response time** for all users is 20 seconds. Usually, it's not that simple. Your application may include a variety of operations with differing characteristics and acceptable response times. You will need to set measurable goals for each of these.

You will also need to determine variances in the load. For example, users might access the system heavily between 9:00am and 10:00am and then again between 1:00pm and 2:00pm, as shown in Figure 1–5. If your peak load occurs on a regular basis, for example, daily or weekly, the conventional wisdom is to configure and tune systems to meet your peak load requirements. The lucky users who access the application in off-time will experience better **response times** than your peak-time users. If your peak load is infrequent, you may be willing to tolerate higher **response times** at peak loads for the cost savings of smaller hardware configurations.

Figure 1–5 Adjusting Capacity and Functional Demand



Factors in Improving Performance

Performance spans several areas:

- Application design: Designing applications that efficiently utilize hardware resources and handle increasing numbers of users effectively.
- Sizing and configuration: Determining the type of hardware needed to support your performance goals. See Chapter 3, "Sizing and Configuration".
- Parameter tuning: Setting configurable parameters to achieve the best performance for your application. See Chapter 5, "Optimizing Apache JServ" and Chapter 4, "Optimizing HTTP Server Performance".
- Performance monitoring: Determining what hardware resources are being used by your application and what **response time** your users are experiencing. See Chapter 2, "Monitoring Your Web Server".
- Troubleshooting: Diagnosing why an application is using excessive hardware resources, or why the **response time** exceeds the desired limit.

See Also:

- Chapter 3, "Sizing and Configuration", for more information on sizing and configuration
- Chapter 4, "Optimizing HTTP Server Performance", and Chapter 5, "Optimizing Apache JServ", for more information on parameter tuning
- Chapter 2, "Monitoring Your Web Server", for more information on performance monitoring

Architecture

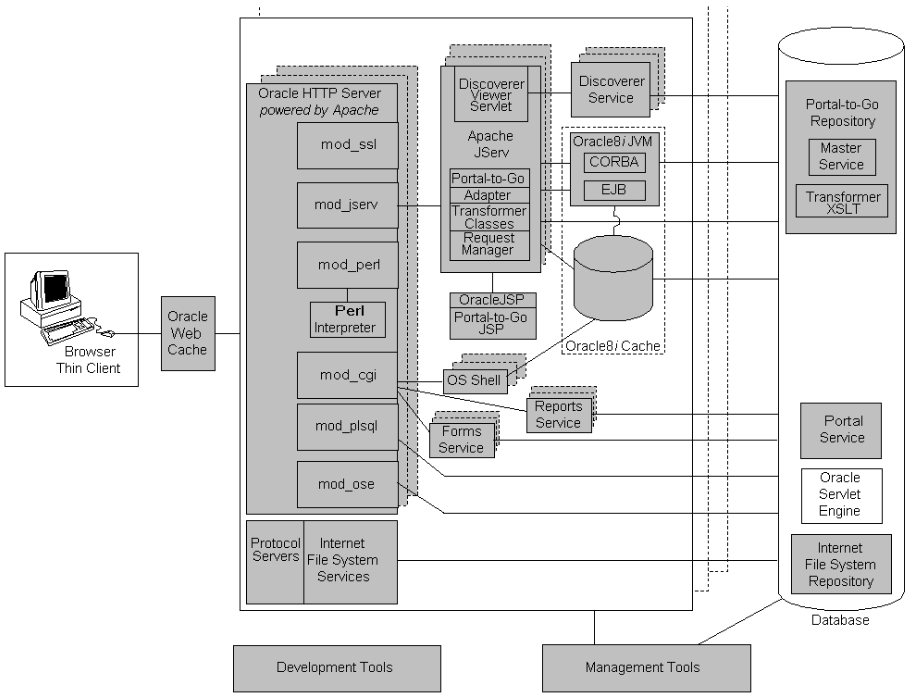
Figure 1–6 shows the architecture of Oracle9*i* Application Server.

This guide addresses the performance and configuration of these components:

- Oracle HTTP Server *powered by Apache*
- Apache JServ
- OracleJSP

See Also: The Oracle9*i* Application Server *Overview Guide* for a list of publications that describe other components.

Figure 1–6 Oracle9i Application Server architecture



2 Monitoring Your Web Server

This chapter explains how to gather performance information from your system. This information helps you to determine the best use of your resources.

This chapter contains the following sections:

- Monitoring Network Activity
- Collecting Performance Data with the Performance Monitor
- Monitoring the Web Server
- Monitoring JServ Processes

Monitoring Network Activity

You can monitor network traffic using the Network Monitor. The Network Monitor must be installed on the Windows NT Server, and the Network Monitor Agent must be installed on the workstation (client) that is to be monitored. The Network Monitor tracks and analyzes network packets transmitted between the two computers.

For information on installing and using the Network Monitor, see the Microsoft website.

Collecting Performance Data with the Performance Monitor

The Performance Monitor is a Windows utility that gathers performance statistics from your operating system and the Oracle HTTP Server. You can use it to:

- Determine resource usage
- Identify performance bottlenecks
- Display statistics from the current activity, or a log file
- Observe the effects of configuration changes on performance

Performance Monitor consumes a small amount of system resources; the amount depends on the frequency, size and location of the data being collected. On average, Performance Monitor uses 2-5 MB of memory and 1-5% CPU time.

The components you can monitor, such as physical disk, logical disk, and memory, are called **objects** in Performance Monitor. Each **object** has its own set of **counters**, performance indicators specific to the **object**. For example, to monitor the HTTP Server or a Java process, you would select the Process **object** and **counters** of interest, such as `% Processor Time`, `% User Time`, `Page Faults/sec`, and `Working Set`.

You can configure any number or combination of objects to monitor. Every system has the following objects:

- System
- Memory
- Cache
- Physical disk
- Logical disk

- Paging file
- Process
- Thread
- Server
- Processor
- Network **objects**, such as the browser and server

Starting the Performance Monitor

To start the Performance Monitor utility:

1. From the Start menu, select Select Programs.
2. From the Administrative Tools menu, select Performance Monitor.

   The Performance Monitor window opens.

Creating a chart of Process Activity

The Performance Monitor Chart view displays the performance counter values in real time, on a strip chart.

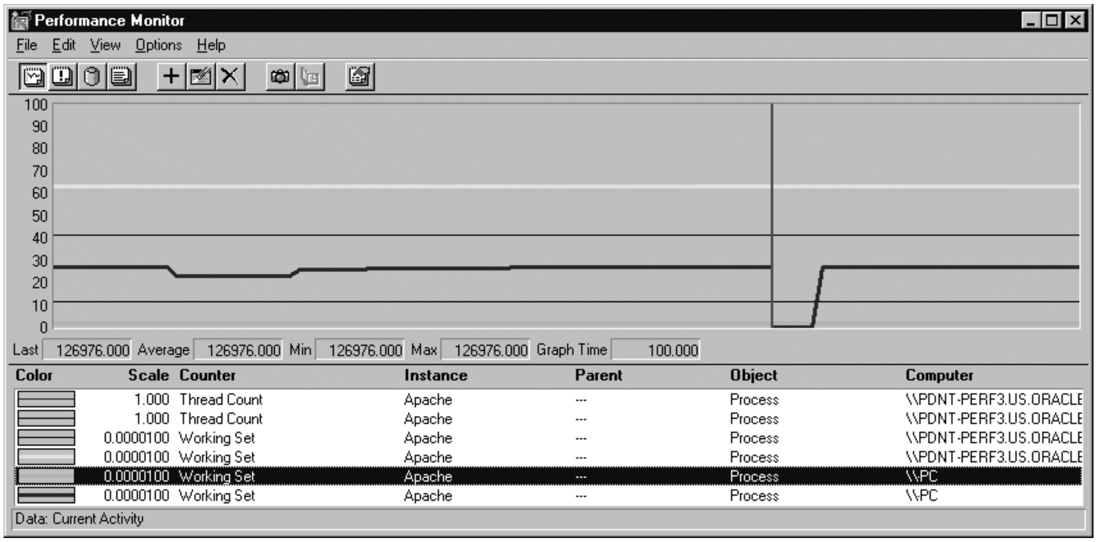
To create a chart of process activity:

1. From the View menu, select Chart.
2. From the Edit menu, select Add to Chart.

   The Add to Chart dialog box opens.
3. Enter or select the hostname of the computer to monitor in the Computer field. (The default is the local computer.)
4. From the Object drop-down list, select Process.
5. From the Instance list, select a **process**. (To monitor the HTTP Server, select the second Apache process. This is the child process; it contains the threads that handle requests).
6. From the Counter list, select the **counters** you want. (To select multiple **counters**, hold down the Ctrl key as you click the counter name.)
7. Click Add.
8. Click Done.

The Performance Monitor window opens with the objects and counters you selected. Figure 2–1 shows a chart view of HTTP Server (Apache) processes on two computers.

Figure 2–1 Performance Monitor chart view



Last 126976.000 Average 126976.000 Min 126976.000 Max 126976.000 Graph Time 100.000

Color	Scale	Counter	Instance	Parent	Object	Computer
▭	1.000	Thread Count	Apache	---	Process	\\PDNT-PERF3.US.ORACLE
▭	1.000	Thread Count	Apache	---	Process	\\PDNT-PERF3.US.ORACLE
▭	0.0000100	Working Set	Apache	---	Process	\\PDNT-PERF3.US.ORACLE
▭	0.0000100	Working Set	Apache	---	Process	\\PDNT-PERF3.US.ORACLE
▭	0.0000100	Working Set	Apache	---	Process	\\PC
▭	0.0000100	Working Set	Apache	---	Process	\\PC

Data: Current Activity

Logging Performance Statistics

The chart view displays the performance statistics in real time, but you can enable logging to save them to a log file.

To enable logging:

1. From the View menu, select Log.
2. From the Edit menu, select Add to Log.

   The Add to Log dialog box opens.
3. Enter or select the hostname of the computer to monitor in the Computer field.
4. Select the objects to monitor. (To select multiple objects, hold down the Ctrl key as you click the object name.)
5. Click Add.
6. Click Done.

7. From the Options menu, select Log.
8. Select the path and enter the filename of the log file to use.
9. Select the Update Time and Interval.
10. Click Start Log.

> **Note:** Because the Performance Monitor log files grow quickly, watch the file size and change to another log file before it becomes unmanageably large. Performance Monitor has no mechanism to monitor file sizes.

Creating a Report or Chart of Log File Data

You can view logged performance data in chart or report format. This procedure assumes that Performance Monitor is running, with a log file status of `Collecting` (you have to stop the log before you can access the log file).

To create a report or chart from a log file you have saved from a prior logging session, start with Step 3.

To select data from the log file:

1. From the Options menu, select Log.
2. Click Stop Log.

   The display area of the window is cleared and the status changes to Closed.
3. From the View menu, select Chart or Report.
4. From the Options menu, select Data From.

   The Data From dialog box appears.
5. Click the Log File radio button, and use the browse button to navigate to your log file.

   The Open Input Log File dialog box opens.
6. Select your file and click Open.
7. Click OK.
8. From the Edit menu, select Time Window.

   The Input Log File Timeframe dialog box opens.

9. Specify the start and stop times of the interval of time you are interested in, using bookmarks or the scroll bar.
10. Click OK.
11. From the Edit menu, select Add to Chart or Add to Report and select the objects and counters to display.
12. Click Add.
13. Click Done.

    The data from the selected time period appears in the chart or report.

Monitoring the Web Server

Monitoring activity on the system is essential to performance tuning. The Oracle HTTP Server provides server side status information, including current server statistics, via the `mod_status` module. To obtain these server status reports, you must configure the web server as described in the following sections.

Using the mod_status Utility to Monitor the Web Server

To enable monitoring, edit the httpd.conf file to replace *your_domain.com* with the hostname of the computer from which you want to monitor.

```
<Location /server-status>
     SetHandler server-status
     Order deny, allow
     Deny from all
     Allow from your_domain.com
</Location>
```

Ensure that the `ExtendedStatus` directive is set to `On`, so that the maximum amount of information is displayed.

When you allow access from all domains, instead of just *your_domain.com*, you can monitor the server from machines outside of your domain, but be aware of the security implications of this: your server status is accessible from any site. It is probably best to specify the domain(s) from which you want to monitor your system.

With monitoring enabled, you can view current statistics from `http://`*`hostname:port`*`/server-status` where hostname:port is the hostname and port you want to monitor. These statistics help you to gain insight on how busy your system is.

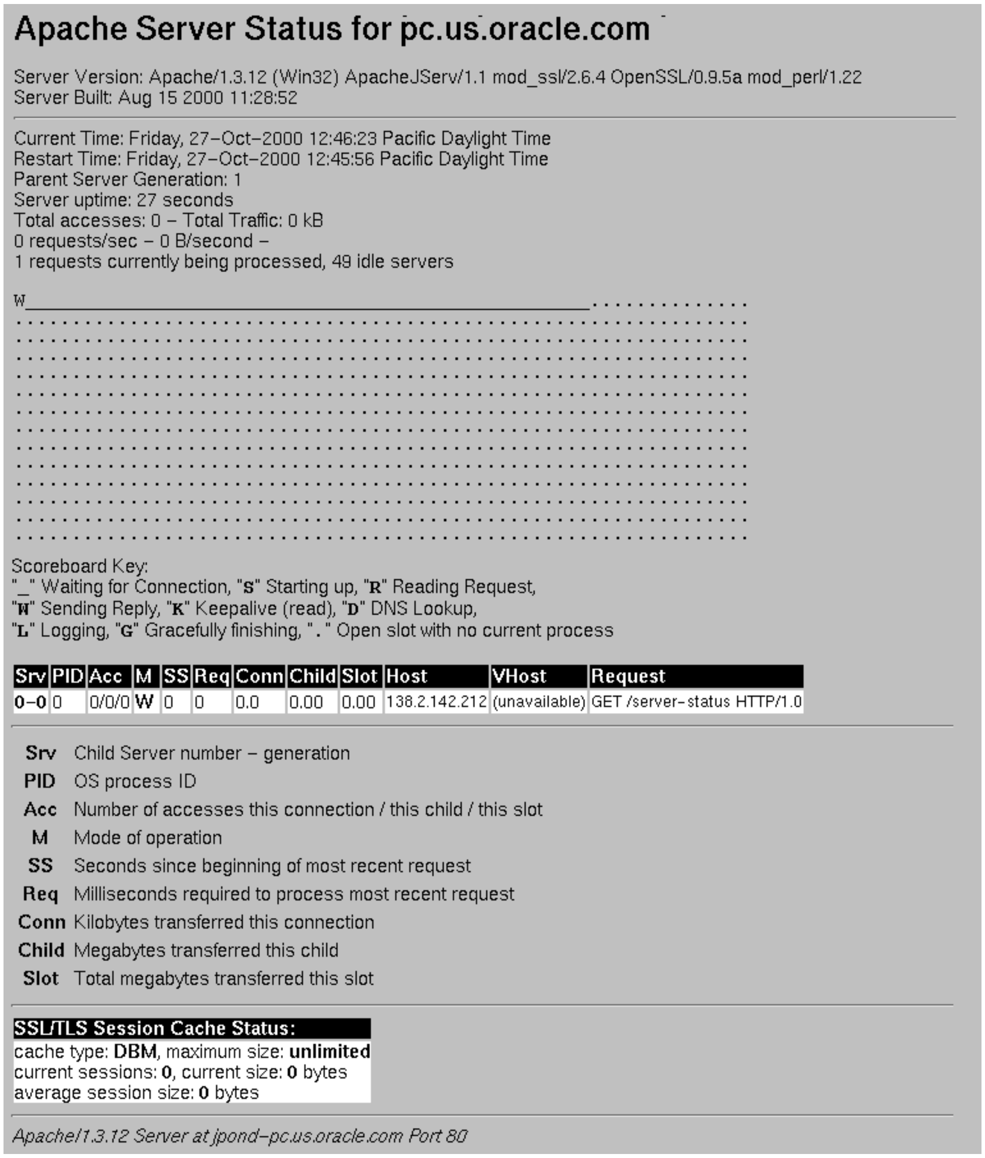
The display includes:

- Hostname for which status is displayed
- Server version
- Date server was built
- Current time, restart time, uptime
- Number of requests currently being processed
- Number of idle servers

- Current server state (e.g., waiting for connection, reading request, sending reply, etc.

Figure 2–2 is a screen capture of a server status page with `ExtendedStatus` turned on.

Figure 2–2 Server status page



Apache Server Status for pc.us.oracle.com

Server Version: Apache/1.3.12 (Win32) ApacheJServ/1.1 mod_ssl/2.6.4 OpenSSL/0.9.5a mod_perl/1.22
Server Built: Aug 15 2000 11:28:52

Current Time: Friday, 27-Oct-2000 12:46:23 Pacific Daylight Time
Restart Time: Friday, 27-Oct-2000 12:45:56 Pacific Daylight Time
Parent Server Generation: 1
Server uptime: 27 seconds
Total accesses: 0 - Total Traffic: 0 kB
0 requests/sec - 0 B/second -
1 requests currently being processed, 49 idle servers

```
W_________________________________________________..............
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
```

Scoreboard Key:
"_" Waiting for Connection, "S" Starting up, "R" Reading Request,
"W" Sending Reply, "K" Keepalive (read), "D" DNS Lookup,
"L" Logging, "G" Gracefully finishing, "." Open slot with no current process

Srv	PID	Acc	M	SS	Req	Conn	Child	Slot	Host	VHost	Request
0-0	0	0/0/0	W	0	0	0.0	0.00	0.00	138.2.142.212	(unavailable)	GET /server-status HTTP/1.0

Srv	Child Server number - generation
PID	OS process ID
Acc	Number of accesses this connection / this child / this slot
M	Mode of operation
SS	Seconds since beginning of most recent request
Req	Milliseconds required to process most recent request
Conn	Kilobytes transferred this connection
Child	Megabytes transferred this child
Slot	Total megabytes transferred this slot

SSL/TLS Session Cache Status:
cache type: **DBM**, maximum size: **unlimited**
current sessions: **0**, current size: **0** bytes
average session size: **0** bytes

Apache/1.3.12 Server at jpond-pc.us.oracle.com Port 80

Interpreting Server Status Information

The display (with `ExtendedStatus` enabled) shows that 1 server is sending a reply. `ThreadsPerChild` is set to 50, so there are 49 idle servers (the busy server is responding to the server-status request). You can determine what stage of processing each server is in from the value in the M (Mode column).

Customizing the Server Status display

Figure 2–2 is a snapshot of a server for a moment in time. You can get updated server statistics at any interval you choose by including the refresh parameter in the server-status URL:

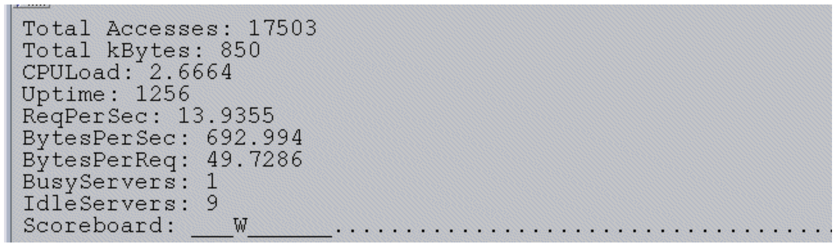
`http://`*`servername:port`*`/server-status?refresh=`*`x`*

where servername:port is the name of the server and port number you are monitoring, and x is an integer representing the number of seconds after which the data is refreshed. For example, specify `refresh=3` to update statistics every 3 seconds.

You may also find it useful to have the statistics displayed in a machine-readable format, for processing in a data analysis or spreadsheet program. To do this, add `auto` to the end of the URL, as shown below:

`http://`*`servername:port`*`/server-status?auto`

Figure 2–3 Server statistics display



```
Total Accesses: 17503
Total kBytes: 850
CPULoad: 2.6664
Uptime: 1256
ReqPerSec: 13.9355
BytesPerSec: 692.994
BytesPerReq: 49.7286
BusyServers: 1
IdleServers: 9
Scoreboard: ___W______....................................
```

Monitoring JServ Processes

After you start the Oracle9*i* Application Server, you can check to ensure that all JServ processes have started normally. If performance is degraded during operation, you can quickly determine if this is because JServ processes have terminated by looking at the Status column (each configured process has a status of `Up` or `Down`).

1. Remove the comments in the JServ status handler section of the `jserv.conf` file to enable monitoring and specify the host(s) that can access JServ status (the default is `localhost`). Be aware of security implications when selecting the hosts that will be allowed to access status information on your system.

   ```
   <Location /jserv/>
   SetHandler jserv-status
      order deny, allow
      deny from all
      allow from hostname_1.com
      allow from hostname_2.com
   </Location>
   ```

2. Type the following into your browser:

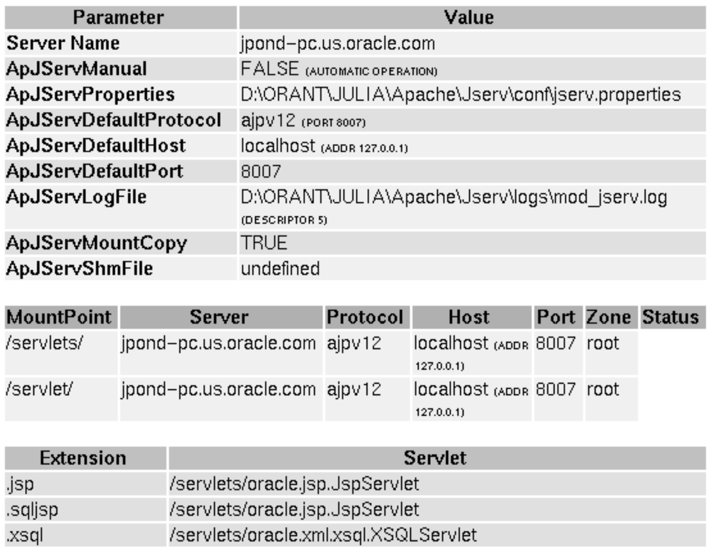
   ```
   http://hostname:port/jserv/
   ```

   The port must be the port on which the web server listens (found in the `httpd.conf` file). A Configured Hosts column displays links to hosts.

3. Click the host to monitor.

   The JServ status information for the host displays as shown in Figure 2–4.

Figure 2–4   JServ status display



Parameter	Value
Server Name	jpond–pc.us.oracle.com
ApJServManual	FALSE (AUTOMATIC OPERATION)
ApJServProperties	D:\ORANT\JULIA\Apache\Jserv\conf\jserv.properties
ApJServDefaultProtocol	ajpv12 (PORT 8007)
ApJServDefaultHost	localhost (ADDR 127.0.0.1)
ApJServDefaultPort	8007
ApJServLogFile	D:\ORANT\JULIA\Apache\Jserv\logs\mod_jserv.log (DESCRIPTOR 5)
ApJServMountCopy	TRUE
ApJServShmFile	undefined

MountPoint	Server	Protocol	Host	Port	Zone	Status
/servlets/	jpond–pc.us.oracle.com	ajpv12	localhost (ADDR 127.0.0.1)	8007	root	
/servlet/	jpond–pc.us.oracle.com	ajpv12	localhost (ADDR 127.0.0.1)	8007	root	

Extension	Servlet
.jsp	/servlets/oracle.jsp.JspServlet
.sqljsp	/servlets/oracle.jsp.JspServlet
.xsql	/servlets/oracle.xml.xsql.XSQLServlet

3 Sizing and Configuration

This chapter provides guidelines for sizing and configuration which can help you meet performance goals. It also discusses performance factors such as CPU and memory consumption.

This chapter contains the following sections:

Sizing your Hardware and Resources

In addition to the minimum installation recommendations, your hardware resources need to be adequate for the requirements of your specific applications. To avoid hardware-related performance bottlenecks, each hardware component should operate at no more than 80% of capacity.

Processor and memory resources in particular should be allocated generously, for the maximum user load expected.

Understanding Concurrent Users and User Population

The amount of hardware resources required varies based on the application. A common mistake is to use resource estimates that do not incorporate user **think time** and network latencies. In sizing applications, you must have some idea of the relationship between the number of potential users and the number of concurrent users. This is determined by the **think time** and the average **response time** for your application.

To determine memory requirements, you also need to consider the number of concurrent executing users (not the total user population) times the cost per user.

Note: The `ThreadsPerChild` setting in your `httpd.conf` file limits the number of concurrently executing users.

Table 3–1 provides an example of the impact of **think time** and **service time** on the **concurrency** and resulting performance of a system.

Table 3–1 Concurrent executing users

User population[1]	Think time (sec)[2]	Service time (sec)[3]	Range of concurrent users[4]	Average response Time (sec)[5]	Requests per second (throughput)[6]	CPU utilization (%)[7]
100	0	0.3	100	5.2	19	99
100	1	0.3	65-100	4.2	19	99
100	10	0.3	0-32	0.9	9	48
100	10	0.6	0-53	2.9	8	80

[1] User population - total users.

[2] **Think time** - the time the user is not engaged in actual use of the processor (the time between requests).

[3] **Service time** (seconds) - elapsed time to complete the operation measured for a single user.

[4] Range of concurrent users - the number of users measured on the server, taken in snapshots from the `server-status` display (requests currently being processed). See "Using the mod_status Utility to Monitor the Web Server" on page 2-7 for information on `server-status`.

[5] Average **response time** - **response time** measured at the client under load.

[6] Requests per second (**throughput**) - number of requests processed.

[7] CPU utilization - average total CPU utilization as a percentage.

Determining CPU Requirements

For most applications, the majority of the CPU utilization is spent in processing the application’s code. The CPU requirement of any application depends on its complexity and workload, as shown in Table 3–2.

You will need to monitor the CPU requirements of applications throughout the development cycle. See Chapter 2, "Monitoring Your Web Server" for information on how to do this.

Table 3–2 Application CPU requirements on a 400 MHz x86 processor

Application	CPU requirement (per request)
Static page, 20KB	5 msec
Simple servlet, JDK (Java Developer’s Kit) release 1.2	50 msec

Determining Memory Requirements

This section discusses the following memory requirements:

- Determining Memory Requirements for the Oracle HTTP Server
- Determining Memory Requirements for JServ
- Determining Java Heap Size
- Determining Memory Requirements for Servlets and OracleJSP pages
- Determining the Number of JServ Processes per CPU

Determining Memory Requirements for the Oracle HTTP Server

The parent HTTP server process consumes up to 6 MB in the resident set. The child process, which handles the requests, consumed up to 12 MB in in-house tests.

Determining Memory Requirements for JServ

A JServ process using JDK 1.2 requires about 7 MB in the resident set at startup. The process will grow to use up to 12MB physical memory. While the memory required may grow to be much larger, it is important to note that NT limits the amount that can be kept in the working set in physical memory. If the process grows quite large, then there may be a large number of page faults in the JServ process, and the paging that results will cause reduced throughput and increased response times. If there is sufficient memory on the host, however, the process will be allowed to cache pages in the system file cache, thereby reducing the necessity to fetch pages from disk. The system file cache memory will be reclaimed by the system if necessary. This means that the most important counter to watch for in the Apache and Jserv processes is the number of page faults (see the Processes tab in the Task Manager). If this is consistently higher than 5 faults per second, and there is little or no system memory available (see the Performance tab in the Task Manager), then adding physical memory is likely to improve performance.

Determining Java Heap Size

For JDK release 1.2, the default maximum heap size is 67 MB.

To maximize performance, set the maximum heap size to accommodate application requirements. To determine how much Java heap you need, include calls in your program to the `Runtime.getRuntime().totalMemory()` and `Runtime.getRuntime().freeMemory` methods in the `java.lang` package.

Subtract free memory from total memory; the difference is the amount of heap that the application consumed.

Suppose you determine that you need 128MB of heap. To change the heap size, you would set the maximum Java heap size in the `jserv.properties` file for automatic mode:

```
wrapper.bin.parameters=-mx128m
```

In manual mode, if more than one JServ process is running, the heap size must be set on the command line or in the startup script for each JServ process.

When a JServ process exceeds its maximum heap size, the process terminates. In automatic mode, a new process is started, but performance is degraded significantly. In manual mode, a terminated process will not be restarted, so ensure that the heap size is sufficient.

Determining Memory Requirements for Servlets and OracleJSP pages

In general, OracleJSP pages require more memory than servlets. In in-house tests using JDK release 1.2, a servlet and an OracleJSP required just under 7 MB at startup. In the OracleJSP, the JServ process size continued to grow up to about 12 MB, and then a large number of page faults occurred, decreasing throughput and increasing the average response time.

See Also: "Determining Memory Requirements for JServ" on page 3-4.

In the servlet, the JServ process size for a simple "Hello, World" servlet stabilized at 8 MB. The amount of memory needed also depends on whether sessions are being used. With sessions turned off, the OracleJSP page and the servlet stabilized at around 8 MB. You can turn sessions off by including the following line at the top of the page:

```
<%@ page session="false" %>
```

You should also set developer mode to `false` in the `jserv.properties` file.

Determining the Number of JServ Processes per CPU

Oracle recommends two JServ processesper CPU as a starting point. In-house tests on a server with four CPUs and one JServ process produced many failed requests under load. With two JServ processes, there were occasional errors, but the response time increased dramatically as the load increased.

Using four JServ processes, performance was acceptable. Under heavy load, with 8 JServ processes and the number of concurrent requests per JServ process limited to 10, response time was reduced by 50% and throughput increased.

To limit the number of concurrent requests per JServ process, change the `security.maxConnections` parameter in the `jserv.properties` file. This parameter specifies the maximum number of JServ requests that can be handled simultaneously. The default setting of 50 results in exponential increases in response time under load, unless the overall incoming load is restricted (by reducing the `ThreadsPerChild` value in the `httpd.conf` file). Oracle recommends that you limit `security.maxConnections` to 10, to avoid synchronization bottlenecks in the JServ processes.

Note: If your servlets implement the SingleThreadModel interface, you must reduce `security.maxConnections` to a value less than the `singleThreadModelServlet.maximumCapacity` value (the default is 10).

Unless a significant number of the requests on your HTTP server are non-JServ, ensure that the following is true:

(number of JServ processes) x (`security.maxConnections`) = (the `ThreadsPerChild` value) x 2

Otherwise, under high loads, requests will fail due to connection failures between the HTTP server and the JServ engine. The HTTP server passes the request to the next JServ process in the list, but because the security.maxConnections value is exceeded, response time is degraded significantly in waiting for a connection.

If your application code performs a lot of synchronization, or creates many new Java objects, then you should consider increasing the number of JServ processes, while limiting the number of threads per process to between 10 and 20. In this way you avoid increased queuing and processing required for object synchronization in the JVM (Java virtual machine). This is because the httpd (mod_jserv) process sends incoming requests to the JServ processes in a distributed fashion.

> **See Also:** "How to Perform Load Balancing" on page 5-4 for instructions on changing parameters in the configuration files, and details on how requests are distributed among the available JServ engines.

4 Optimizing HTTP Server Performance

This chapter provides information on improving the Oracle HTTP Server's performance, including tuning network parameters, the effects of changing the `ThreadsPerChild` parameter, and the performance impacts of logging.

This chapter contains the following sections:

Network Tuning

There are a number of things to keep in mind when running the Oracle HTTP Server.

1. Be certain that you have sufficient memory. System memory usage can be monitored by watching the display under the performance tab in the Task Manager.
2. Be certain that only the TPC/IP protocol stack is running. If another protocol is running, it will be listed in the list under the Protocols tab of the Control Panel/Network dialog box. To remove it, select it with the mouse, and click Remove. If you close the Network dialog box, you will be prompted to restart the system. It will be easier, however, to first continue with step 3.
3. Select the "Maximize Throughput for File Sharing" network optimization scheme. Under the Services tab of the Control Panel/Network dialog box, you can examine the Server properties. Select "Server" in the list, and click Properties. This will bring up a dialog box that allows you to choose the criteria for which TCP will be optimized. The default setting is "Maximize Throughput for File Sharing". We recommend you use the setting. If this has been otherwise set, reset it to the default and click "OK". Then close the Control Panel/Network dialog box. If you changed this setting, you will be prompted to restart the system, and if you have made any changes in Step 1 above, or in this box, you should do so.

Note: The performance is much better when either "Maximize Throughput for File Sharing" or "Maximize Throughput for Network Applications" is chosen, than when either of the other options is chosen. We have also see that the response time under load is cut in half when we maximize for file sharing rather than for network applications.

In addition to the above, one can adjust individual TCP/IP parameters in the registry. We do not recommend that you do so as it is complex. Unless you have plenty of time to test the impact for your environment, we recommend you limit your TCP/IP tuning to the steps above.

Configuring the ThreadsPerChild Parameter

The `ThreadsPerChild` parameter in the `httpd.conf` file specifies the number of requests that can be handled concurrently by the HTTP server. Requests in excess of the `ThreadsPerChild` parameter value wait in the TCP/IP queue. Allowing the requests to wait in the TCP/IP queue often results in the best **response time** and **throughput**.

Configuring ThreadsPerChild for Servlet Requests

If the HTTP server will handle servlets exclusively, then the ThreadsPerChild parameter value must be much smaller than the number of concurrent requests that the JServ process can service. If a JServ process is handling all the requests it can (as specified by `security.maxConnections` in the `jserv.properties` file), then a delay occurs when the HTTP server tries to establish a connection to it.

To prevent the corresponding increase in **latency**, set `ThreadsPerChild` to half the number of requests that all of the JServ processes can handle. For example, suppose you have four JServ processes, and each has a `security.maxConnections` value of 10. The total number of requests that the JServ processes can handle is 40, so set `ThreadsPerChild` to 20.

Configuring ThreadsPerChild for Static Page Requests

The more concurrent threads you make available to handle requests, the more requests your server can process. But be aware that with too many threads, under high load, requests will be handled more slowly and the server will consume more system resources.

In in-house tests of static page requests, a setting of 20 `ThreadsPerChild` per CPU produced good **response time** and **throughput** results. For example, if you have four CPUs, set `ThreadsPerChild` to 80. If, with this setting, CPU utilization does not exceed 85%, you can increase `ThreadsPerChild`, but ensure that the available threads are in use. You can determine this using the `mod_status` utility.

> **See Also:** "Using the mod_status Utility to Monitor the Web Server" on page 2-7 for information.

Enabling SSL Session Caching

The Oracle HTTP server caches a client's SSL session information by default. With session caching, only the first connection to the server incurs high **latency**. For example, in a simple test to connect and disconnect to an SSL-enabled server, the elapsed time for 5 connections was 5.54 seconds without SSL session caching. With SSL session caching enabled, the elapsed time for 5 round trips was 1.18 seconds.

The `SSLSessionCacheTimeout` directive in `httpd.conf` determines how long the server keeps a session alive (the default is `300` seconds). The session information is kept in a file. You can specify where to keep the session information using the `SSLSessionCache` directive; the default location is the `%ORACLE_HOME%\Apache\Apache\logs\` directory. The file can be used by multiple Oracle HTTP Server processes.

The duration of an SSL session is unrelated to the use of HTTP persistent connections.

Understanding Performance Implications of Logging

This section discusses the performance implications of using access logging and the `HostNameLookups` directive.

Access Logging

For static page requests, access logging of the default fields results in a 2-7% performance cost.

HostNameLookups

By default, the `HostNameLookups` directive is set to `Off`. The server writes the IP addresses of incoming requests to the log files. When `HostNameLookups` is set to on, the server queries the DNS system on the Internet to find the host name associated with the IP address of each request, then writes the host names to the log.

Performance degraded by about 10-12% (best case) in Oracle in-house tests with `HostNameLookups` set to on. Depending on the server load and the network connectivity to your DNS server, the performance cost of the DNS lookup could be high. Unless you really need to have host names in your logs in real time, it is best to log IP addresses.

Benefits of the HTTP/1.1 Protocol

The Oracle HTTP server can use HTTP / 1.1. Netscape Navigator release 4.0 still uses HTTP / 1.0, with some 1.1 features, such as persistent connections. Internet Explorer uses HTTP / 1.1. The performance benefit of persistent connections comes from reducing the overhead of establishing and tearing down a connection for each request. A persistent connection accepts multiple requests from a user.

For a small static page request, the connection **latency** can equal or exceed the response **latency** (the time to fulfill the request after the connection is established), so using persistent connections can result in major performance gains.

Supporting Persistent Connections

If your users' browsers support persistent connections (the default behavior of HTTP / 1.1), you can support them on the server using the `KeepAlive` directives in the Oracle HTTP Server. (Some browsers that do not support all HTTP / 1.1 features do support persistent connections; for example, recent versions of Netscape.)

How Persistent Connections Improve Response Times

Persistent connections can improve total **response time** for a web interaction that involves multiple HTTP requests, because the delay of setting up a connection only happens once.

Consider the total time required, without persistent connections, for a client to retrieve a web page with three images from the server.

Activity	Seconds
Establish connection	1
Produce and send the text portion of the page	5
Establish connection	1
Transfer first image file	2
Establish connection	1
Transfer second image file	2
Establish connection	1
Transfer third image file	2
Total	15

With persistent connections, the **response time** for the same request is reduced:

Activity	Seconds
Establish connection	1
Produce and send the text portion of the page	5
Transfer first image file	2
Transfer second image file	2
Transfer third image file	2
Total	12

This is a 20% reduction in **service time**.

How Persistent Connections Reduce Server Workload

Another benefit of persistent connections is reduction of the work load on the server. Because the server need not repeat the work to set up the connection with a client, it is free to perform other work.

In-house tests using an OracleJSP application (`lotto.jsp`, one of the samples that ships with Oracle9*i* Application Server) and persistent connections showed an improvement of about 20%, with a single user making 5 requests per connection. With an increased number of users (10-100), the performance improvement was less dramatic, but still significant (6% or better).

5 Optimizing Apache JServ

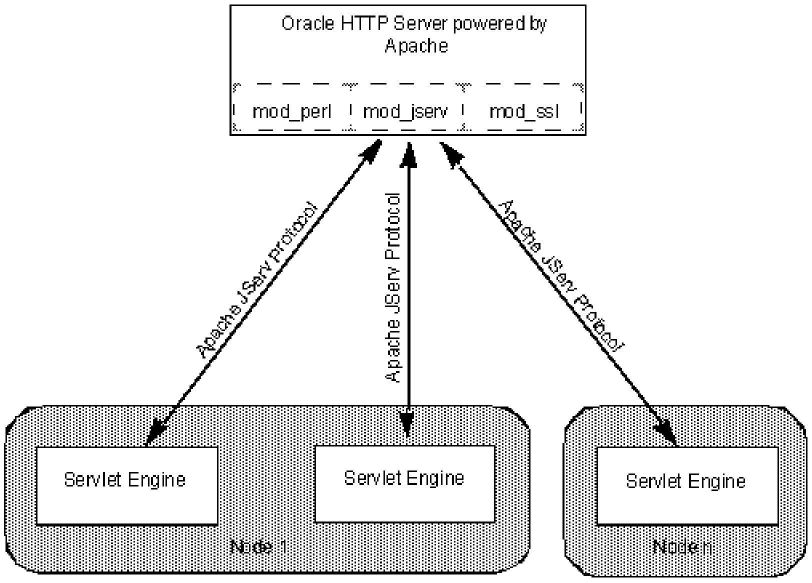
This chapter describes the JServ architecture, and discusses ways you can improve JServ performance. It also includes performance information on OracleJSP pages (the Oracle implementation of Sun Microsystems' JavaServer Pages 1.1.)

Overview of JServ

Apache JServ is made up of an Apache module called `mod_jserv`, which runs in the httpd process, and a servlet engine, which runs in a Java process. `mod_jserv`, which is implemented in C, functions as a dispatcher, routing each servlet request to a JServ process for execution.

The servlet engine runs in its own JVM (Java Virtual Machine) and is solely responsible for parsing the request and generating a response. As Figure 5–1 shows, multiple JServs can service requests. The HTTP server process and the JServ process communicate using the Apache JServ Protocol 1.2.

Figure 5–1 Apache JServ components



Optimizing Servlet Performance

This section discusses strategies for optimizing JServ performance: loading servlets when starting the JVM, and load balancing.

The terms “repository” and “zone” are used in this discussion. Servlets, repositories, and zones are analogous to files, directories and virtual hosts. A servlet is a single unit, a **repository** is a collection of servlets, and a **zone** is a collection of repositories.

Loading Servlet Classes

Apache JServ allows you to load servlet classes when the JVM is started. To do this, put the servlets to load in the `servlets.startup` directive in the servlet zone properties file. When the servlet is loaded, its `init()` method is called. All other servlets (those not listed in `servlets.startup`) are loaded and initialized on first request.

Using this facility increases the start-up time for your JServ process, but improves first-request **latency** for servlets.

Pre-Loading with JSPs

If you are using a JSP as the servlet (your code does not extend `HttpServlet`), you will be unable to use this pre-load option, but you could pre-load the JSP runner by including the `oracle.jsp.jspServlet` in `servlets.startup`.

If the first-request **latency** for your initialization routines is really a performance issue, you can achieve some of the results described above by creating a dummy servlet to call your one-time initialization routines in its `init()` method. You must add the name of the dummy servlet to `servlets.startup`.

Reloading Servlet Classes Automatically

If `autoreload.classes` is set to `true` for a zone (the default), then each time one of that zone’s servlets is requested, every class that has been loaded from a repository in that zone is checked to see if it has been modified. If one of the classes has changed, then all previously loaded classes from the zone’s repositories are unloaded, which means that as the classes are needed, they will be loaded from their class files again.

This is a useful development feature, because you can install new versions or drop in new class files without restarting the server. For optimal performance in production environments, however, you should set both automatic class reloading parameters to false, since there is a performance cost in checking the repositories on every execution of a servlet. Change these parameters in the zone properties file:

```
autoreload.classes=false
autoreload.file=false
```

How to Perform Load Balancing

It is often beneficial to spread the servlet application load among multiple JServ processes, especially when the application is run on a multiprocessor system or if the servlets and HTTP server are run on separate nodes. Running multiple Apache JServ processes generally results in higher **throughput** and shorter **response time**, even on a single-processor host. (See Chapter 3, "Sizing and Configuration" for specific recommendations.)

This section explains how to balance incoming requests between two JServ processes running on the same host as the HTTP server. Examples from the `jserv.properties` files are included with the procedures; substitute your own port numbers and directory locations where needed.

With this method of load balancing, you must start and stop processes manually, because JServ cannot automatically start and stop more than one JServ process. (Sample scripts for starting and stopping the JServ processes and the Oracle HTTP Server are included in the `%ORACLE_HOME%\Apache\Apache\bin\` directory.) This means that if a process terminates for any reason, JServ will not restart it. To prevent processes from terminating due to memory shortage, ensure that you have a sufficient maximum heap size set for your JServ processes. See "Determining Java Heap Size" on page 3-4.

Configuring the JServ processes

Each JServ process in your load balancing scheme must be configured to listen on its own port and to log to its own file. If you have a `jserv.properties` file containing the parameters needed to run your application, you can duplicate it to create a properties file for each JServ process.

1. Create a properties file for each JServ process.

   ```
   copy jserv.properties jserv1.properties
   copy jserv.properties jserv2.properties
   ```

2. Edit *`jserv1.properties`* as follows:

   ```
   port=8001
   log.file=E:\Oracle\iSuites\Apache\jserv\logs\jserv1.log
   ```

3. Edit `jserv2.properties` as follows:

   ```
   port=8002
   log.file=E:\Oracle\iSuites\Apache\jserv\logs\jserv2.log
   ```

   > **Note:** If your HTTP server will be running on a different host than the JServ processes, you must also add the IP address of the host running the HTTP server to the `security.allowedAddresses` parameter in each `jserv.properties` file.

To start and stop the processes and the web server, it is convenient to use scripts. Samples are included in the `%ORACLE_HOME%\Apache\Apache\bin\` directory (`startJServ.bat` and `stopJServ.bat`).

Modifying jserv.conf to distribute the load

1. Set the flag to start processes manually.

   ```
   ApJServManual on
   ```

2. Indicate where the servlet request is to be sent.

   a. Locate the ApJServMount directive.

      ```
      ApJServMount /servlets /root
      ```

      If the user requests `http://`*`your.server.com`*`/servlets/`*`testServlet`*, the ApJServMount directive above will execute `testServlet` in the **zone** called `/root`.

   b. Change the **zone** identifier from `/root` to `balance://Jserv_set/root` and then add the ApJServBalance, ApJServHost, ApJservRoute directives for each process sharing the load, as shown below:

      ```
      ApJServMount /servlets balance://JServ_set/root
      ApJServBalance JServ_set JServ1
      ApJServBalance JServ_set JServ2 2
      ApJServHost JServ1 ajpv12://127.0.0.1:8001
      ApJServHost JServ2 ajpv12://127.0.0.1:8002
      ApJServRoute JS1 JServ1
      ApJServRoute JS2 JServ2
      ```

* The `ApJServMount` directive, with `/servlets balance://JServ_set/root`, now balances requests for servlets in `/servlets` between `JServ1` and `JServ2`.
* The `ApJServBalance` directive identifies `JServ1` and `JServ2` as the processes that share the load. The ’`2`’ following `JServ2` is a weight value. It specifies that twice as many requests will be sent to `JServ2` as would be otherwise, i.e., that `JServ2` will get about 2/3 of all incoming requests. See "Distribution of JServ Requests" below for details.
* The `ApJServHost` directive identifies the host and port on which the processes are listening.
* The `ApJServRoute` directive associates JServ processes with sessions. JServ uses this information to keep all of a session’s requests together in one process. The JServ session mechanism sends the process route information back to the user (generally in a cookie). You need only modify it if your application uses sessions.

Distribution of JServ Requests

The following process explains how `mod_jserv` selects the JServ engine to handle a request:

1. The httpd process is started.
2. `mod_jserv` creates a list of available JServs, with extra entries for JServs with a weight value greater than `1` (for example, JServ2 in the example above, as specified by `ApJServBalance Jserv_set JServ2 2`).
3. The httpd daemon receives a servlet request and hands it to `mod_jserv`.
4. `mod_jserv` selects the JServ engine that will handle the request.
   a. `mod_jserv` checks to see if the request is part of a current session. If so, it uses the `ApJServRoute` directives to find the JServ that handled the other requests for that session.
   b. If the request is not part of a session, `mod_jserv` selects the next JServ process in its list (round robin request distribution).

Using Single Thread Model Servlets

Oracle recommends that you write your servlets to implement the SingleThreadModel (STM) interface. An application that was modified to implement the STM interface demonstrated a 25% improvement in **response time**.

It is also much easier to manage database connections with STM servlets. The database connection can be set up in the `init()` method of the servlet, and closed in the `destroy()` method. When executing the servlet's `doGet()` or `service()` method, you need not be concerned with obtaining a database connection. You can also manage database connections with JDBC connection caching.

There are three parameters in the `zone.properties` file that impact the performance of STM servlets in particular. These govern:

- The minimum number of servlet object instances that will be generated and available after the servlet class is loaded
- The maximum number that can be generated
- The number that should be generated if the available instances are insufficient

Because it is very costly to generate instances while the system is running, Oracle recommends that you set your minimum to equal your maximum value. The optimum value depends somewhat on how many connections your database server can handle. This should be split among the JServ processes, as follows:

Total DB connections / Number of JServ processes = Number of STM servlet instances per process

See Chapter 3, "Sizing and Configuration" for suggestions on determining the right number of JServ processes for your application, and "How to Perform Load Balancing" on page 5-4 for the steps to configure them. Suppose you've determined that you want 10 servlet instances per process. The capacity settings in the `zone.properties` file would be:

```
singleThreadModelServlet.initialCapacity = 10
singleThreadModelServlet.incrementCapacity = 0
singleThreadModelServlet.maximumCapacity = 10
```

Warning: The value for `singleThreadModelServlet.maximumCapacity` **in the zone properties file must be at least as large as the value for security.maxConnections in the jserv.properties file. If it is not, and the number of requests sent to the JServ process exceeds the maximum capacity, requests will fail.**

What is OracleJSP?

OracleJSP 1.1.0.0 is Oracle's implementation of the Sun Microsystems JavaServer Pages 1.1 specification. Some of the additional features it includes are custom JavaBeans for accessing Oracle databases, SQL support, and extended data types. See the *Oracle9i Application Server Overview Guide* in the Oracle9*i* Application Server documentation library for detailed descriptions of the features.

Tuning OracleJSP Pages for Performance

This section explains how you can improve OracleJSP pages' performance.

Impact of Session Management

In general, sessions add performance overhead; they consume about 0.5 KB of resident memory. You must turn off sessions if you do not want a new session to be created with each request. By default, sessions are enabled in OracleJSP pages, so if they are not being used, turn them off by including the following line at the top of the page:

```
<%@ page session="false" %>
```

If you are going to use sessions, ensure that you explicitly close them. If you don't, they will linger until they time out (the default value for session timeout is 30 minutes). To close a session manually, use the `session.invalidate()` method.

See the *OracleJSP Developer's Guide and Reference* in the Oracle9*i* Application Server documentation library for more information on configuring OracleJSP pages.

Developer Mode

Another parameter that has a significant effect on performance is developer mode. It is a useful feature for debugging during development, but it degrades performance. The default value is true, so you will need to set it to false in the `jserv.properties` file as follows:

```
servlet.oracle.jsp.JspServlet.initArgs=developer_mode=false
```

With developer mode set to true, OracleJSP and the servlet engine examines every request to determine whether to reload or retranslate the page or application. With developer mode off, only the first request is examined.

In a test using JDK 1.2 with 50 users, 128 MB heap, and the default TCP settings, the performance gains with developer mode off were 14% in **throughput**, and 28% in average **response time**.

Buffering

If an OracleJSP page is not using any features that do not require resetting the buffer (such as error pages, `contextType` settings, forwards, etc.), disabling the JSP page buffer will improve performance. This is because memory will not be used in creating the buffer, and the output can go directly to the browser. Use this page directive to disable buffering:

```
<%@ page buffer=”none” %>
```

The default size of an OracleJSP page buffer is 8 KB.

OracleJSP Performance Tips

The configuration actions below can enhance the performance of your OracleJSP pages.

Caching database connections

Since the performance cost of creating database connections is high, it is more performant to use a cache of connections. If you use a cache of database connections, then the OracleJSP application can get a connection from the cache and return it when it is finished.

Configuring the statement batch value

The JDBC driver accumulates a number of execution requests (the batch value) and passes them to the database to be processed at the same time. You can configure the batch value to control how frequently processing occurs.

Caching JDBC statements

Cache executable statements that are repeatedly used, to avoid re-parsing, statement object re-creation, and recalculation of parameter size definitions.

Pre-fetching rows

During a query, pre-fetch multiple rows into the client to reduce round trips between the database and the server.

Caching rowsets from the database

Cache small sets of data that are accessed frequently and do not change often. This is not as beneficial for large data sets, since they consume more memory.

Invoking static includes

To invoke static includes, use the page directive:

```
<%@ include file="/jsp/filename.jsp” %>
```

Static include creates a copy of the file in the JSP, thereby affecting its page size. This is useful in avoiding trips to the request dispatcher (unlike dynamic includes, which must go through the request dispatcher each time). However, file sizes should be small to avoid exceeding the 64 KB limit of the service method of the generated page implementation class.

Invoking Dynamic Includes

To invoke dynamic includes, use the page directive

```
<jsp:include page=”/jsp/filename.jsp” flush="true" />
```

This directive is analogous to a function call, and therefore does not increase the page size of the JSP. However, a dynamic include increases the processing overhead since it must go through the request dispatcher. Dynamic includes are useful for including other pages without increasing page size.

Index