Eclipse GlassFish Server Performance Tuning Guide, Release 5.1

Table of Contents

Eclipse GlassFish Server

Performance Tuning Guide

Release 5.1

Contributed 2018, 2019

This book describes how to get the best performance with GlassFish Server 5.1.

Preface

This documentation is part of the Java Enterprise Edition contribution to the Eclipse Foundation and is not intended for use in relation to Java Enterprise Edition or Orace GlassFish. The documentation is in the process of being revised to reflect the new Jakarta EE branding. Additional changes will be made as requirements and procedures evolve for Jakarta EE. Where applicable, references to Java EE or Java Enterprise Edition should be considered references to Jakarta EE.

Please see the Title page for additional license information.

The Performance Tuning Guide describes how to get the best performance with GlassFish Server 5.1.

This preface contains information about and conventions for the entire Eclipse GlassFish Server (GlassFish Server) documentation set.

GlassFish Server 5.1 is developed through the GlassFish project open-source community at `http://glassfish.java.net/`. The GlassFish project provides a structured process for developing the GlassFish Server platform that makes the new features of the Java EE platform available faster, while maintaining the most important feature of Java EE: compatibility. It enables Java developers to access the GlassFish Server source code and to contribute to the development of the GlassFish Server. The GlassFish project is designed to encourage communication between Oracle engineers and the community.

Oracle GlassFish Server Documentation Set

Book Title	Description
Release Notes	Provides late-breaking information about the software and the documentation and includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.
Quick Start Guide	Explains how to get started with the GlassFish Server product.
Installation Guide	Explains how to install the software and its components.
Upgrade Guide	Explains how to upgrade to the latest version of GlassFish Server. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Deployment Planning Guide	Explains how to build a production deployment of GlassFish Server that meets the requirements of your system and enterprise.

Book Title	Description
Administration Guide	Explains how to configure, monitor, and manage GlassFish Server subsystems and components from the command line by using the `asadmin` utility. Instructions for performing these tasks from the Administration Console are provided in the Administration Console online help.
Security Guide	Provides instructions for configuring and administering GlassFish Server security.
Application Deployment Guide	Explains how to assemble and deploy applications to the GlassFish Server and provides information about deployment descriptors.
Application Development Guide	Explains how to create and implement Java Platform, Enterprise Edition (Java EE platform) applications that are intended to run on the GlassFish Server. These applications follow the open Java standards model for Java EE components and application programmer interfaces (APIs). This guide provides information about developer tools, security, and debugging.
Embedded Server Guide	Explains how to run applications in embedded GlassFish Server and to develop applications in which GlassFish Server is embedded.
High Availability Administration Guide	Explains how to configure GlassFish Server to provide higher availability and scalability through failover and load balancing.
Performance Tuning Guide	Explains how to optimize the performance of GlassFish Server.
Troubleshooting Guide	Describes common problems that you might encounter when using GlassFish Server and explains how to solve them.
Error Message Reference	Describes error messages that you might encounter when using GlassFish Server.
Reference Manual	Provides reference information in man page format for GlassFish Server administration commands, utility commands, and related concepts.
Message Queue Release Notes	Describes new features, compatibility issues, and existing bugs for Open Message Queue.
Message Queue Technical Overview	Provides an introduction to the technology, concepts, architecture, capabilities, and features of the Message Queue messaging service.
Message Queue Administration Guide	Explains how to set up and manage a Message Queue messaging system.
Message Queue Developer's Guide for JMX Clients	Describes the application programming interface in Message Queue for programmatically configuring and monitoring Message Queue resources in conformance with the Java Management Extensions (JMX).
Message Queue Developer's Guide for Java Clients	Provides information about concepts and procedures for developing Java messaging applications (Java clients) that work with GlassFish Server.

Book Title	Description
Message Queue Developer's Guide for C Clients	Provides programming and reference information for developers working with Message Queue who want to use the C language binding to the Message Queue messaging service to send, receive, and process Message Queue messages.

Typographic Conventions

The following table describes the typographic changes that are used in this book.

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name% su`
		
		`Password:`

AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` filename.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the User's Guide. A cache is a copy that is stored locally. Do not save the file.

Symbol Conventions

The following table explains symbols that might be used in this book.

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.

Symbol	Description	Example	Meaning
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
>	Indicates menu item selection in a graphical user interface.	File > New > Templates	From the File menu, choose New. From the New submenu, choose Templates.

Default Paths and File Names

The following table describes the default paths and file names that are used in this book.

Placeholder	Description	Default Value
as-install	Represents the base installation directory for GlassFish Server. In configuration files, as-install is represented as follows: `${com.sun.aas.installRoot}`	Installations on the Oracle Solaris operating system, Linux operating system, and Mac OS operating system: user's-home-directory`/glassfish3/glassfish` Installations on the Windows operating system: SystemDrive`:\glassfish3\glassfish`
as-install-parent	Represents the parent of the base installation directory for GlassFish Server.	Installations on the Oracle Solaris operating system, Linux operating system, and Mac operating system: user's-home-directory`/glassfish3` Installations on the Windows operating system: SystemDrive`:\glassfish3`
domain-root-dir	Represents the directory in which a domain is created by default.	as-install`/domains/`

Placeholder	Description	Default Value
domain-dir	Represents the directory in which a domain's configuration is stored. In configuration files, domain-dir is represented as follows: `${com.sun.aas.instanceRoot}`	domain-root-dir`/`domain-name
instance-dir	Represents the directory for a server instance.	domain-dir`/`instance-name

1 Overview of GlassFish Server Performance Tuning

You can significantly improve performance of the Oracle GlassFish Server and of applications deployed to it by adjusting a few deployment and server configuration settings. However, it is important to understand the environment and performance goals. An optimal configuration for a production environment might not be optimal for a development environment.

The following topics are addressed here:

Process Overview

The following table outlines the overall GlassFish Server 4.0 administration process, and shows where performance tuning fits in the sequence.

Table 1-1 Performance Tuning Roadmap

Step	Description of Task	Location of Instructions
1	Design: Decide on the high-availability topology and set up GlassFish Server.	GlassFish Server Open Source Edition Deployment Planning Guide
2	Capacity Planning: Make sure the systems have sufficient resources to perform well.	GlassFish Server Open Source Edition Deployment Planning Guide
3	Installation: Configure your DAS, clusters, and clustered server instances.	GlassFish Server Open Source Edition Installation Guide

Step	Description of Task	Location of Instructions
4	Deployment: Install and run your applications. Familiarize yourself with how to configure and administer the GlassFish Server.	The following books: • GlassFish Server Open Source Edition Application Deployment Guide • GlassFish Server Open Source Edition Administration Guide
5	High Availability Configuration: Configuring your DAS, clusters, and clustered server instances for high availability and failover	GlassFish Server Open Source Edition High Availability Administration Guide
6	Performance Tuning: Tune the following items: • Applications • GlassFish Server • Java Runtime System • Operating system and platform	The following chapters: • Tuning Your Application • Tuning the GlassFish Server • Tuning the Java Runtime System • Tuning the Operating System and Platform

Performance Tuning Sequence

Application developers should tune applications prior to production use. Tuning applications often produces dramatic performance improvements. System administrators perform the remaining steps in the following list after tuning the application, or when application tuning has to wait and you want to improve performance as much as possible in the meantime.

Ideally, follow this sequence of steps when you are tuning performance:

1. Tune your application, described in Tuning Your Application.
2. Tune the server, described in Tuning the GlassFish Server.
3. Tune the Java runtime system, described in Tuning the Java Runtime System.
4. Tune the operating system, described in Tuning the Operating System and Platform.

Understanding Operational Requirements

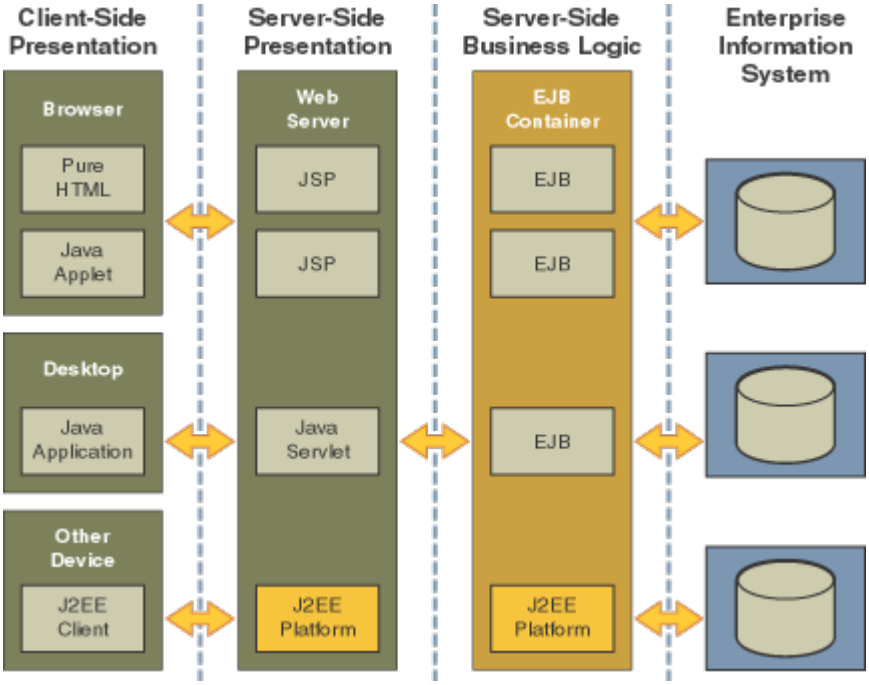
Before you begin to deploy and tune your application on the GlassFish Server, it is important to clearly define the operational environment. The operational environment is determined by high-level constraints and requirements such as:

- Application Architecture
- Security Requirements
- High Availability Clustering, Load Balancing, and Failover
- Hardware Resources
- Administration

Application Architecture

The Java EE Application model, as shown in the following figure, is very flexible; allowing the application architect to split application logic functionally into many tiers. The presentation layer is typically implemented using servlets and JSP technology and executes in the web container.

Figure 1-1 Java EE Application Model



Moderately complex enterprise applications can be developed entirely using servlets and JSP technology. More complex business applications often use Enterprise JavaBeans (EJB) components. The GlassFish Server integrates the Web and EJB containers in a single process. Local access to EJB components from servlets is very efficient. However, some application deployments may require EJB

components to execute in a separate process; and be accessible from standalone client applications as well as servlets. Based on the application architecture, the server administrator can employ the GlassFish Server in multiple tiers, or simply host both the presentation and business logic on a single tier.

It is important to understand the application architecture before designing a new GlassFish Server deployment, and when deploying a new business application to an existing application server deployment.

Security Requirements

Most business applications require security. This section discusses security considerations and decisions.

User Authentication and Authorization

Application users must be authenticated. The GlassFish Server provides a number of choices for user authentication, including file-based, administration, LDAP, certificate, JDBC, digest, PAM, Solaris, and custom realms.

The default file based security realm is suitable for developer environments, where new applications are developed and tested. At deployment time, the server administrator can choose between the Lighweight Directory Access Protocol (LDAP) or Solaris security realms. Many large enterprises use LDAP-based directory servers to maintain employee and customer profiles. Small to medium enterprises that do not already use a directory server may find it advantageous to leverage investment in Solaris security infrastructure.

For more information on security realms, see "Administering Authentication Realms" in GlassFish Server Open Source Edition Security Guide.

The type of authentication mechanism chosen may require additional hardware for the deployment. Typically a directory server executes on a separate server, and may also require a backup for replication and high availability. Refer to the Oracle Java System Directory Server (`http://www.oracle.com/us/products/middleware/identity-management/oracle-directory-services/index.html`) documentation for more information on deployment, sizing, and availability guidelines.

An authenticated user's access to application functions may also need authorization checks. If the application uses the role-based Java EE authorization checks, the application server performs some additional checking, which incurs additional overheads. When you perform capacity planning, you must take this additional overhead into account.

Encryption

For security reasons, sensitive user inputs and application output must be encrypted. Most business-oriented web applications encrypt all or some of the communication flow between the browser and GlassFish Server. Online shopping applications encrypt traffic when the user is completing a purchase or supplying private data. Portal applications such as news and media typically do not employ encryption. Secure Sockets Layer (SSL) is the most common security framework, and is supported by many browsers and application servers.

The GlassFish Server supports SSL 2.0 and 3.0 and contains software support for various cipher suites. It also supports integration of hardware encryption cards for even higher performance. Security considerations, particularly when using the integrated software encryption, will impact hardware sizing and capacity planning.

Consider the following when assessing the encryption needs for a deployment:

- What is the nature of the applications with respect to security? Do they encrypt all or only a part of the application inputs and output? What percentage of the information needs to be securely transmitted?
- Are the applications going to be deployed on an application server that is directly connected to the Internet? Will a web server exist in a demilitarized zone (DMZ) separate from the application server tier and backend enterprise systems?
  A DMZ-style deployment is recommended for high security. It is also useful when the application has a significant amount of static text and image content and some business logic that executes on the GlassFish Server, behind the most secure firewall. GlassFish Server provides secure reverse proxy plugins to enable integration with popular web servers. The GlassFish Server can also be deployed and used as a web server in DMZ.
- Is encryption required between the web servers in the DMZ and application servers in the next tier? The reverse proxy plugins supplied with GlassFish Server support SSL encryption between the web server and application server tier. If SSL is enabled, hardware capacity planning must be take into account the encryption policy and mechanisms.
- If software encryption is to be employed:
  - What is the expected performance overhead for every tier in the system, given the security requirements?
  - What are the performance and throughput characteristics of various choices?

For information on how to encrypt the communication between web servers and GlassFish Server, see "Administering Message Security" in GlassFish Server Open Source Edition Security Guide.

High Availability Clustering, Load Balancing, and Failover

GlassFish Server 4.0 enables multiple GlassFish Server instances to be clustered to provide high

availability through failure protection, scalability, and load balancing.

High availability applications and services provide their functionality continuously, regardless of hardware and software failures. To make such reliability possible, GlassFish Server 4.0 provides mechanisms for maintaining application state data between clustered GlassFish Server instances. Application state data, such as HTTP session data, stateful EJB sessions, and dynamic cache information, is replicated in real time across server instances. If any one server instance goes down, the session state is available to the next failover server, resulting in minimum application downtime and enhanced transactional security.

GlassFish Server provides the following high availability features:

- High Availability Session Persistence
- High Availability Java Message Service
- RMI-IIOP Load Balancing and Failover

See Tuning High Availability Persistence for high availability persistence tuning recommendations.

See the GlassFish Server Open Source Edition High Availability Administration Guide for complete information about configuring high availability clustering, load balancing, and failover features in GlassFish Server 4.0.

Hardware Resources

The type and quantity of hardware resources available greatly influence performance tuning and site planning.

GlassFish Server provides excellent vertical scalability. It can scale to efficiently utilize multiple high-performance CPUs, using just one application server process. A smaller number of application server instances makes maintenance easier and administration less expensive. Also, deploying several related applications on fewer application servers can improve performance, due to better data locality, and reuse of cached data between co-located applications. Such servers must also contain large amounts of memory, disk space, and network capacity to cope with increased load.

GlassFish Server can also be deployed on large "farms" of relatively modest hardware units. Business applications can be partitioned across various server instances. Using one or more external load balancers can efficiently spread user access across all the application server instances. A horizontal scaling approach may improve availability, lower hardware costs and is suitable for some types of applications. However, this approach requires administration of more application server instances and hardware nodes.

Administration

A single GlassFish Server installation on a server can encompass multiple instances. A group of one or more instances that are administered by a single Administration Server is called a domain. Grouping server instances into domains permits different people to independently administer the groups.

You can use a single-instance domain to create a "sandbox" for a particular developer and environment. In this scenario, each developer administers his or her own application server, without interfering with other application server domains. A small development group may choose to create multiple instances in a shared administrative domain for collaborative development.

In a deployment environment, an administrator can create domains based on application and business function. For example, internal Human Resources applications may be hosted on one or more servers in one Administrative domain, while external customer applications are hosted on several administrative domains in a server farm.

GlassFish Server supports virtual server capability for web applications. For example, a web application hosting service provider can host different URL domains on a single GlassFish Server process for efficient administration.

For detailed information on administration, see the GlassFish Server Open Source Edition Administration Guide.

General Tuning Concepts

Some key concepts that affect performance tuning are:

- User load
- Application scalability
- Margins of safety

The following table describes these concepts, and how they are measured in practice. The left most column describes the general concept, the second column gives the practical ramifications of the concept, the third column describes the measurements, and the right most column describes the value sources.

Table 1-2 Factors That Affect Performance

Concept	In practice	Measurement	Value sources
User Load	Concurrent sessions at peak load	Transactions Per Minute (TPM) Web Interactions Per Second (WIPS)	(Max. number of concurrent users) * (expected response time) / (time between clicks) Example: (100 users * 2 sec) / 10 sec = 20
Application Scalability	Transaction rate measured on one CPU	TPM or WIPS	Measured from workload benchmark. Perform at each tier.
Vertical scalability	Increase in performance from additional CPUs	Percentage gain per additional CPU	Based on curve fitting from benchmark. Perform tests while gradually increasing the number of CPUs. Identify the "knee" of the curve, where additional CPUs are providing uneconomical gains in performance. Requires tuning as described in this guide. Perform at each tier and iterate if necessary. Stop here if this meets performance requirements.
Horizontal scalability	Increase in performance from additional servers	Percentage gain per additional server process and/or hardware node.	Use a well-tuned single application server instance, as in previous step. Measure how much each additional server instance and hardware node improves performance.
Safety Margins	High availability requirements	If the system must cope with failures, size the system to meet performance requirements assuming that one or more application server instances are non functional	Different equations used if high availability is required.
+	Excess capacity for unexpected peaks	It is desirable to operate a server at less than its benchmarked peak, for some safety margin	80% system capacity utilization at peak loads may work for most installations. Measure your deployment under real and simulated peak loads.

Capacity Planning

The previous discussion guides you towards defining a deployment architecture. However, you determine the actual size of the deployment by a process called capacity planning. Capacity planning enables you to predict:

- The performance capacity of a particular hardware configuration.
- The hardware resources required to sustain specified application load and performance.

You can estimate these values through careful performance benchmarking, using an application with realistic data sets and workloads.

To Determine Capacity

1. Determine performance on a single CPU.
   First determine the largest load that a single processor can sustain. You can obtain this figure by measuring the performance of the application on a single-processor machine. Either leverage the performance numbers of an existing application with similar processing characteristics or, ideally, use the actual application and workload in a testing environment. Make sure that the application and data resources are tiered exactly as they would be in the final deployment.
2. Determine vertical scalability.

Determine how much additional performance you gain when you add processors. That is, you are indirectly measuring the amount of shared resource contention that occurs on the server for a specific workload. Either obtain this information based on additional load testing of the application on a multiprocessor system, or leverage existing information from a similar application that has already been load tested.

Running a series of performance tests on one to eight CPUs, in incremental steps, generally provides a sense of the vertical scalability characteristics of the system. Be sure to properly tune the application, GlassFish Server, backend database resources, and operating system so that they do not skew the results. 3. Determine horizontal scalability.
If sufficiently powerful hardware resources are available, a single hardware node may meet the performance requirements. However for better availability, you can cluster two or more systems. Employing external load balancers and workload simulation, determine the performance benefits of replicating one well-tuned application server node, as determined in step 2.

User Expectations

Application end-users generally have some performance expectations. Often you can numerically quantify them. To ensure that customer needs are met, you must understand these expectations clearly, and use them in capacity planning.

Consider the following questions regarding performance expectations:

- What do users expect the average response times to be for various interactions with the application? What are the most frequent interactions? Are there any extremely time-critical interactions? What is the length of each transaction, including think time? In many cases, you may

need to perform empirical user studies to get good estimates.

- What are the anticipated steady-state and peak user loads? Are there are any particular times of the day, week, or year when you observe or expect to observe load peaks? While there may be several million registered customers for an online business, at any one time only a fraction of them are logged in and performing business transactions. A common mistake during capacity planning is to use the total size of customer population as the basis and not the average and peak numbers for concurrent users. The number of concurrent users also may exhibit patterns over time.
- What is the average and peak amount of data transferred per request? This value is also application-specific. Good estimates for content size, combined with other usage patterns, will help you anticipate network capacity needs.
- What is the expected growth in user load over the next year? Planning ahead for the future will help avoid crisis situations and system downtimes for upgrades.

Further Information

- For more information on Java performance, see Java Performance Documentation (`http://java.sun.com/docs/performance`) and Java Performance BluePrints (`http://java.sun.com/blueprints/performance/index.html`).
- For more information about performance tuning for high availability configurations, see the GlassFish Server Open Source Edition High Availability Administration Guide.
- For complete information about using the Performance Tuning features available through the GlassFish Server Administration Console, refer to the Administration Console online help.
- For details on optimizing EJB components, see Seven Rules for Optimizing Entity Beans (`http://java.sun.com/developer/technicalArticles/ebeans/sevenrules/`)
- For details on profiling, see "Profiling Tools" in GlassFish Server Open Source Edition Application Development Guide.
- To view a demonstration video showing how to use the GlassFish Server Performance Tuner, see the Oracle GlassFish Server 3.1 - Performance Tuner demo (`http://www.youtube.com/watch?v=FavsE2pzAjc`).
- To find additional Performance Tuning development information, see the Performance Tuner in Oracle GlassFish Server 3.1 (`http://blogs.oracle.com/jenblog/entry/performance_tuner_in_oracle_glassfish`) blog.

2 Tuning Your Application

This chapter provides information on tuning applications for maximum performance. A complete guide to writing high performance Java and Java EE applications is beyond the scope of this document.

The following topics are addressed here:

- Java Programming Guidelines
- Java Server Page and Servlet Tuning
- EJB Performance Tuning

Java Programming Guidelines

This section covers issues related to Java coding and performance. The guidelines outlined are not specific to GlassFish Server, but are general rules that are useful in many situations. For a complete discussion of Java coding best practices, see the Java Blueprints (`http://www.oracle.com/technetwork/java/javaee/blueprints/index.html`).

The following topics are addressed here:

- Avoid Serialization and Deserialization
- Use `StringBuilder` to Concatenate Strings
- Assign null to Variables That Are No Longer Needed
- Declare Methods as final Only If Necessary
- Declare Constants as static final
- Avoid Finalizers
- Declare Method Arguments final
- Synchronize Only When Necessary
- Use DataHandlers for SOAP Attachments

Avoid Serialization and Deserialization

Serialization and deserialization of objects is a CPU-intensive procedure and is likely to slow down your application. Use the `transient` keyword to reduce the amount of data serialized. Additionally, customized `readObject()` and `writeObject()` methods may be beneficial in some cases.

Use `StringBuilder` to Concatenate Strings

To improve performance, instead of using string concatenation, use `StringBuilder.append()`.

String objects are immutable - that is, they never change after creation. For example, consider the following code:

```
String str = "testing";
str = str + "abc";
str = str + "def";
```

The compiler translates this code as:

```
String str = "testing";
StringBuilder tmp = new StringBuilder(str);
tmp.append("abc");
str = tmp.toString();
StringBulder tmp = new StringBuilder(str);
tmp.append("def");
str = tmp.toString();
```

This copying is inherently expensive and overusing it can reduce performance significantly. You are far better off writing:

```
StringBuilder tmp = new StringBuilder("testing");
tmp.append("abc");
tmp.append("def");
String str = tmp.toString();
```

Assign null to Variables That Are No Longer Needed

Explicitly assigning a null value to variables that are no longer needed helps the garbage collector to identify the parts of memory that can be safely reclaimed. Although Java provides memory management, it does not prevent memory leaks or using excessive amounts of memory.

An application may induce memory leaks by not releasing object references. Doing so prevents the Java garbage collector from reclaiming those objects, and results in increasing amounts of memory being used. Explicitly nullifying references to variables after their use allows the garbage collector to reclaim memory.

One way to detect memory leaks is to employ profiling tools and take memory snapshots after each transaction. A leak-free application in steady state will show a steady active heap memory after

garbage collections.

Declare Methods as final Only If Necessary

Modern optimizing dynamic compilers can perform inlining and other inter-procedural optimizations, even if Java methods are not declared `final`. Use the keyword `final` as it was originally intended: for program architecture reasons and maintainability.

Only if you are absolutely certain that a method must not be overridden, use the `final` keyword.

Declare Constants as static final

The dynamic compiler can perform some constant folding optimizations easily, when you declare constants as `static final` variables.

Avoid Finalizers

Adding finalizers to code makes the garbage collector more expensive and unpredictable. The virtual machine does not guarantee the time at which finalizers are run. Finalizers may not always be executed, before the program exits. Releasing critical resources in `finalize()` methods may lead to unpredictable application behavior.

Declare Method Arguments final

Declare method arguments `final` if they are not modified in the method. In general, declare all variables `final` if they are not modified after being initialized or set to some value.

Synchronize Only When Necessary

Do not synchronize code blocks or methods unless synchronization is required. Keep synchronized blocks or methods as short as possible to avoid scalability bottlenecks. Use the Java Collections Framework for unsynchronized data structures instead of more expensive alternatives such as`java.util.HashTable`.

Use DataHandlers for SOAP Attachments

Using a `javax.activation.DataHandler` for a SOAP attachment will improve performance.

JAX-RPC specifies:

- A mapping of certain MIME types to Java types.
- Any MIME type is mappable to a `javax.activation.DataHandler` .

As a result, send an attachment (`.gif` or XML document) as a SOAP attachment to an RPC style web service by utilizing the Java type mappings. When passing in any of the mandated Java type mappings (appropriate for the attachment's MIME type) as an argument for the web service, the JAX-RPC runtime handles these as SOAP attachments.

For example, to send out an `image/gif` attachment, use `java.awt.Image`, or create a `DataHandler` wrapper over your image. The advantages of using the wrapper are:

- Reduced coding: You can reuse generic attachment code to handle the attachments because the `DataHandler` determines the content type of the contained data automatically. This feature is especially useful when using a document style service. Since the content is known at runtime, there is no need to make calls to `attachment.setContent(stringContent, "image/gif")`, for example.
- Improved Performance: Informal tests have shown that using `DataHandler` wrappers doubles throughput for `image/gif` MIME types, and multiplies throughput by approximately 1.5 for `text/xml` or `java.awt.Image` for `image/*` types.

Java Server Page and Servlet Tuning

Many applications running on the GlassFish Server use servlets or JavaServer Pages (JSP) technology in the presentation tier. This section describes how to improve performance of such applications, both through coding practices and through deployment and configuration settings.

Suggested Coding Practices

This section provides some tips on coding practices that improve servlet and JSP application performance.

The following topics are addressed here:

- General Guidelines
- Avoid Shared Modified Class Variables
- HTTP Session Handling

- Configuration and Deployment Tips

General Guidelines

Follow these general guidelines to increase performance of the presentation tier:

- Minimize Java synchronization in servlets.
- Do not use the single thread model for servlets.
- Use the servlet's `init()` method to perform expensive one-time initialization.
- Avoid using `System.out.println()` calls.

Avoid Shared Modified Class Variables

In the servlet multithread model (the default), a single instance of a servlet is created for each application server instance. All requests for a servlet on that application instance share the same servlet instance. This can lead to thread contention if there are synchronization blocks in the servlet code. Therefore, avoid using shared modified class variables because they create the need for synchronization.

HTTP Session Handling

Follow these guidelines when using HTTP sessions:

- Create sessions sparingly. Session creation is not free. If a session is not required, do not create one.
- Use `javax.servlet.http.HttpSession.invalidate()` to release sessions when they are no longer needed.
- Keep session size small, to reduce response times. If possible, keep session size below 7 kilobytes.
- Use the directive `<%page session="false"%>` in JSP files to prevent the GlassFish Server from automatically creating sessions when they are not necessary.
- Avoid large object graphs in an `HttpSession`. They force serialization and add computational overhead. Generally, do not store large objects as `HttpSession` variables.
- Do not cache transaction data in an `HttpSession`. Access to data in an `HttpSession` is not transactional. Do not use it as a cache of transactional data, which is better kept in the database and accessed using entity beans. Transactions will rollback upon failures to their original state. However, stale and inaccurate data may remain in `HttpSession` objects. GlassFish Server provides "read-only" bean-managed persistence entity beans for cached access to read-only data.

Configuration and Deployment Tips

Follow these configuration tips to improve performance. These tips are intended for production environments, not development environments.

- To improve class loading time, avoid having excessive directories in the server `CLASSPATH`. Put application-related classes into JAR files.
- HTTP response times are dependent on how the keep-alive subsystem and the HTTP server is tuned in general. For more information, see HTTP Service Settings.
- Cache servlet results when possible. For more information, see "Developing Web Applications" in GlassFish Server Open Source Edition Application Development Guide.
- If an application does not contain any EJB components, deploy the application as a WAR file, not an EAR file.

Optimize SSL

Optimize SSL by using routines in the appropriate operating system library for concurrent access to heap space. The library to use depends on the version of the Solaris Operating System (SolarisOS) that you are using. To ensure that you use the correct library, set the `LD_PRELOAD` environment variable to specify the correct library file. For more information, refer to the following table.

Solaris OS Version	Library	Setting of `LD_PRELOAD` Environment Variable
10	`libumem3LIB`	`/usr/lib/libumem.so`
9	`libmtmalloc3LIB`	`/usr/lib/libmtmalloc.so`

To set the `LD_PRELOAD` environment variable, edit the entry for this environment variable in the `startserv` script. The `startserv` script is located is located in the `bin/startserv` directory of your domain.

The exact syntax to define an environment variable depends on the shell that you are using.

Disable Security Manager

The security manager is expensive because calls to required resources must call the `doPrivileged()` method and must also check the resource with the `server.policy` file. If you are sure that no malicious code will be run on the server and you do not use authentication within your application, then you can disable the security manager.

See "Enabling and Disabling the Security Manager" in GlassFish Server Open Source Edition Application Development Guide for instructions on enabling or disabling the security manager. If using the GlassFish Server Administration Console, navigate to the Configurations>configuration-

name>Security node and check or uncheck the Security Manager option as desired. Refer to the Administration Console online help for more information.

EJB Performance Tuning

The GlassFish Server's high-performance EJB container has numerous parameters that affect performance. Individual EJB components also have parameters that affect performance. The value of individual EJB component's parameter overrides the value of the same parameter for the EJB container. The default values are designed for a single-processor computer system. Modify these values as appropriate to optimize for other system configurations.

The following topics are addressed here:

Goals

The goals of EJB performance tuning are:

- Increased speed: Cache as many beans in the EJB caches as possible to increase speed (equivalently, decrease response time). Caching eliminates CPU-intensive operations. However, since memory is finite, as the caches become larger, housekeeping for them (including garbage collection) takes longer.
- Decreased memory consumption: Beans in the pools or caches consume memory from the Java virtual machine heap. Very large pools and caches degrade performance because they require longer and more frequent garbage collection cycles.
- Improved functional properties: Functional properties such as user timeout, commit options, security, and transaction options, are mostly related to the functionality and configuration of the application. Generally, they do not compromise functionality for performance. In some cases, you might be forced to make a "trade-off" decision between functionality and performance. This section

offers suggestions in such cases.

Monitoring EJB Components

When the EJB container has monitoring enabled, you can examine statistics for individual beans based on the bean pool and cache settings.

For example, the monitoring command below returns the Bean Cache statistics for a stateful session bean.

```
asadmin get --user admin --host e4800-241-a --port 4848
-m specjcmp.application.SPECjAppServer.ejb-module.
   supplier_jar.stateful-session-bean.BuyerSes.bean-cache.*
```

The following is a sample of the monitoring output:

```
resize-quantity = -1
 cache-misses = 0
 idle-timeout-in-seconds = 0
 num-passivations = 0
 cache-hits = 59
 num-passivation-errors = 0
 total-beans-in-cache = 59
 num-expired-sessions-removed = 0
 max-beans-in-cache = 4096
 num-passivation-success = 0
```

The monitoring command below gives the bean pool statistics for an entity bean:

```
asadmin get --user admin --host e4800-241-a --port 4848
-m specjcmp.application.SPECjAppServer.ejb-module.
   supplier_jar.stateful-entity-bean.ItemEnt.bean-pool.*
idle-timeout-in-seconds = 0
steady-pool-size = 0
total-beans-destroyed = 0
num-threads-waiting = 0
num-beans-in-pool = 54
max-pool-size = 2147483647
pool-resize-quantity = 0
total-beans-created = 255
```

The monitoring command below gives the bean pool statistics for a stateless bean.

```
asadmin get --user admin --host e4800-241-a --port 4848
-m test.application.testEjbMon.ejb-module.slsb.stateless-session-bean.slsb.bean-pool.*
idle-timeout-in-seconds = 200
steady-pool-size = 32
total-beans-destroyed = 12
num-threads-waiting = 0
num-beans-in-pool = 4
max-pool-size = 1024
pool-resize-quantity = 12
total-beans-created = 42
```

Tuning the bean involves charting the behavior of the cache and pool for the bean in question over a period of time.

If too many passivations are happening and the JVM heap remains fairly small, then the `max-cache-size` or the `cache-idle-timeout-in-seconds` can be increased. If garbage collection is happening too frequently, and the pool size is growing, but the cache hit rate is small, then the `pool-idle-timeout-in-seconds` can be reduced to destroy the instances.

Note:

Specifying a `max-pool-size` of zero (0) means that the pool is unbounded. The pooled beans remain in memory unless they are removed by specifying a small interval for `pool-idle-timeout-in-seconds`. For production systems, specifying the pool as unbounded is NOT recommended.

Monitoring Individual EJB Components

To gather method invocation statistics for all methods in a bean, use the following command:

```
asadmin get -m monitorableObject.*
```

where monitorableObject is a fully-qualified identifier from the hierarchy of objects that can be monitored, shown below.

```
serverInstance.application.applicationName.ejb-module.moduleName
```

where moduleName is `x_jar` for module `x.jar`.

- `.stateless-session-bean.beanName .bean-pool .bean-method.methodName`
- `.stateful-session-bean.beanName .bean-cache .bean-method.methodName`

- `.entity-bean.beanName .bean-cache .bean-pool .bean-method.methodName`
- `.message-driven-bean.beanName .bean-pool .bean-method.methodName (methodName = onMessage)`

For standalone beans, use this pattern:

```
serverInstance.application.applicationName.standalone-ejb-module.moduleName
```

The possible identifiers are the same as for `ejb-module`.

For example, to get statistics for a method in an entity bean, use this command:

```
asadmin get -m serverInstance.application.appName.ejb-module.moduleName
.entity-bean.beanName.bean-method.methodName.*
```

For more information about administering the monitoring service in general, see "Administering the Monitoring Service" in GlassFish Server Open Source Edition Administration Guide. For information about viewing comprehensive EJB monitoring statistics, see "EJB Statistics" in GlassFish Server Open Source Edition Administration Guide.

To configure EJB monitoring using the GlassFish Server Administration Console, navigate to the Configurations>configuration-name>Monitoring node. After configuring monitoring, you can view monitoring statistics by navigating to the server (Admin Server) node and then selecting the Monitor tab. Refer to the Administration Console online help for instructions on each of these procedures.

Alternatively, to list EJB statistics, use the `asadmin list` subcommand. For more information, see `list (1)`.

For statistics on stateful session bean passivations, use this command:

```
asadmin get -m serverInstance.application.appName.ejb-module.moduleName
.stateful-session-bean.beanName.bean-cache.*
```

From the attribute values that are returned, use this command:

`num-passivationsnum-passivation-errorsnum-passivation-success`

General Guidelines

The following guidelines can improve performance of EJB components. Keep in mind that decomposing an application into many EJB components creates overhead and can degrade performance. EJB components are not simply Java objects. They are components with semantics for remote call interfaces, security, and transactions, as well as properties and methods.

Use High Performance Beans

Use high-performance beans as much as possible to improve the overall performance of your application. For more information, see Tuning Tips for Specific Types of EJB Components.

The types of EJB components are listed below, from the highest performance to the lowest:

1. Stateless Session Beans and Message Driven Beans
2. Stateful Session Beans
3. Container Managed Persistence (CMP) entity beans configured as read-only
4. Bean Managed Persistence (BMP) entity beans configured as read-only
5. CMP beans
6. BMP beans

For more information about configuring high availability session persistence, see "Configuring High Availability Session Persistence and Failover" in GlassFish Server Open Source Edition High Availability Administration Guide. To configure EJB beans using the GlassFish Server Administration Console, navigate to the Configurations>configuration-name>EJB Container node and then refer to the Administration Console online help for detailed instructions.

Use Caching

Caching can greatly improve performance when used wisely. For example:

- Cache EJB references: To avoid a JNDI lookup for every request, cache EJB references in servlets.
- Cache home interfaces: Since repeated lookups to a home interface can be expensive, cache references to `EJBHomes` in the `init()` methods of servlets.
- Cache EJB resources: Use `setSessionContext()` or `ejbCreate()` to cache bean resources. This is again an example of using bean lifecycle methods to perform application actions only once where possible. Remember to release acquired resources in the `ejbRemove()` method.

Use the Appropriate Stubs

The stub classes needed by EJB applications are generated dynamically at runtime when an EJB client needs them. This means that it is not necessary to generate the stubs or retrieve the client JAR file when deploying an application with remote EJB components. When deploying an application, it is no longer necessary to specify the `--retrieve` option, which can speed up deployment.

If you have a legacy rich-client application that directly uses the CosNaming service (not a recommended configuration), then you must generate the stubs for your application explicitly using RMIC. For more information, see the GlassFish Server Open Source Edition Troubleshooting Guide for more details.

Remove Unneeded Stateful Session Beans

Removing unneeded stateful session beans avoids passivating them, which requires disk operations.

Cache and Pool Tuning Tips

Follow these tips when using the EJB cache and pools to improve performance:

- Explicitly call `remove()`: Allow stateful session EJB components to be removed from the container cache by explicitly calling of the `remove()` method in the client.
- Tune the entity EJB component's pool size: Entity Beans use both the EJB pool and cache settings. Tune the entity EJB component's pool size to minimize the creation and destruction of beans. Populating the pool with a non-zero steady size before hand is useful for getting better response for initial requests.
- Cache bean-specific resources: Use the `setEntityContext()` method to cache bean specific resources and release them using the `unSetEntityContext()` method.
- Load related data efficiently for container-managed relationships (CMRs). For more information, see Pre-Fetching Container Managed Relationship (CMR) Beans.
- Identify read-only beans: Configure read-only entity beans for read only operations. For more information, see Read-Only Entity Beans.

Using Local and Remote Interfaces

This section describes some considerations when EJB components are used by local and remote clients.

Prefer Local Interfaces

An EJB component can have remote and local interfaces. Clients not located in the same application server instance as the bean (remote clients) use the remote interface to access the bean. Calls to the remote interface require marshalling arguments, transportation of the marshalled data over the network, un-marshaling the arguments, and dispatch at the receiving end. Thus, using the remote interface entails significant overhead.

If an EJB component has a local interface, then local clients in the same application server instance can use it instead of the remote interface. Using the local interface is more efficient, since it does not require argument marshalling, transportation, and un-marshalling.

If a bean is to be used only by local clients then it makes sense to provide only the local interface. If, on the other hand, the bean is to be location-independent, then you should provide both the remote and local interfaces so that remote clients use the remote interface and local clients can use the local interface for efficiency.

Using Pass-By-Reference Semantics

By default, the GlassFish Server uses pass-by-value semantics for calling the remote interface of a bean, even if it is co-located. This can be expensive, since clients using pass-by-value semantics must copy arguments before passing them to the EJB component.

However, local clients can use pass-by-reference semantics and thus the local and remote interfaces can share the passed objects. But this means that the argument objects must be implemented properly, so that they are shareable. In general, it is more efficient to use pass-by-reference semantics when possible.

Using the remote and local interfaces appropriately means that clients can access EJB components efficiently. That is, local clients use the local interface with pass-by-reference semantics, while remote clients use the remote interface with pass-by-value semantics.

However, in some instances it might not be possible to use the local interface, for example when:

- The application predates the EJB 2.0 specification and was written without any local interfaces.
- There are bean-to-bean calls and the client beans are written without making any co-location assumptions about the called beans.

For these cases, the GlassFish Server provides a pass-by-reference option that clients can use to pass arguments by reference to the remote interface of a co-located EJB component.

You can specify the pass-by-reference option for an entire application or a single EJB component. When specified at the application level, all beans in the application use pass-by-reference semantics when passing arguments to their remote interfaces. When specified at the bean level, all calls to the remote interface of the bean use pass-by-reference semantics. See "Value Added Features" in GlassFish Server Open Source Edition Application Development Guide for more details about the pass-by-reference flag.

To specify that an EJB component will use pass by reference semantics, use the following tag in the `sun-ejb-jar.xml` deployment descriptor:

```
<pass-by-reference>true</pass-by-reference>
```

This avoids copying arguments when the EJB component's methods are invoked and avoids copying results when methods return. However, problems will arise if the data is modified by another source during the invocation.

Improving Performance of EJB Transactions

This section provides some tips to improve performance when using transactions.

The following topics are addressed here:

- Use Container-Managed Transactions
- Do Not Encompass User Input Time
- Identify Non-Transactional Methods
- Use `TX_REQUIRED` for Long Transaction Chains
- Use Lowest Cost Database Locking
- Use XA-Capable Data Sources Only When Needed
- Configure JDBC Resources as One-Phase Commit Resources
- Use the Least Expensive Transaction Attribute

Use Container-Managed Transactions

Container-managed transactions are preferred for consistency, and provide better performance.

Do Not Encompass User Input Time

To avoid resources being held unnecessarily for long periods, a transaction should not encompass user input or user think time.

Identify Non-Transactional Methods

Declare non-transactional methods of session EJB components with `NotSupported` or `Never` transaction attributes. These attributes can be found in the `ejb-jar.xml` deployment descriptor file. Transactions should span the minimum time possible since they lock database rows.

Use `TX_REQUIRED` for Long Transaction Chains

For very large transaction chains, use the transaction attribute `TX_REQUIRED`. To ensure EJB methods in a call chain, use the same transaction.

Use Lowest Cost Database Locking

Use the lowest cost locking available from the database that is consistent with any transaction. Commit the data after the transaction completes rather than after each method call.

Use XA-Capable Data Sources Only When Needed

When multiple database resources, connector resources or JMS resources are involved in one transaction, a distributed or global transaction needs to be performed. This requires XA capable resource managers and data sources. Use XA capable data sources, only when two or more data source are going to be involved in a transaction. If a database participates in some distributed transactions, but mostly in local or single database transactions, it is advisable to register two separate JDBC resources and use the appropriate resource in the application.

Configure JDBC Resources as One-Phase Commit Resources

To improve performance of transactions involving multiple resources, the GlassFish Server uses last agent optimization (LAO), which allows the configuration of one of the resources in a distributed transaction as a one-phase commit (1PC) resource. Since the overhead of multiple-resource transactions is much higher for a JDBC resource than a message queue, LAO substantially improves performance of distributed transactions involving one JDBC resource and one or more message queues. To take advantage of LAO, configure a JDBC resource as a 1PC resource. Nothing special needs to be done to configure JMS resources.

In global transactions involving multiple JDBC resources, LAO will still improve performance, however, not as much as for one JDBC resource. In this situation, one of the JDBC resources should be configured as 1PC, and all others should be configured as XA.

Use the Least Expensive Transaction Attribute

Set the following transaction attributes in the EJB deployment descriptor file (`ejb-jar.xml`). Options are listed from best performance to worst. To improve performance, choose the least expensive attribute

that will provide the functionality your application needs:

1. `NEVER`
2. `TX_NOTSUPPORTED`
3. `TX_MANDATORY`
4. `TX_SUPPORTS`
5. `TX_REQUIRED`
6. `TX_REQUIRESNEW`

Using Special Techniques

Special performance-enhancing techniques are discussed in the following sections:

- Version Consistency
- Request Partitioning

Version Consistency

> Note:
>
> The technique in section applies only to the EJB 2.1 architecture. In the EJB 3.0 architecture, use the Java Persistence API (JPA).

Use version consistency to improve performance while protecting the integrity of data in the database. Since the application server can use multiple copies of an EJB component simultaneously, an EJB component's state can potentially become corrupted through simultaneous access.

The standard way of preventing corruption is to lock the database row associated with a particular bean. This prevents the bean from being accessed by two simultaneous transactions and thus protects data. However, it also decreases performance, since it effectively serializes all EJB access.

Version consistency is another approach to protecting EJB data integrity. To use version consistency, you specify a column in the database to use as a version number. The EJB lifecycle then proceeds like this:

- The first time the bean is used, the `ejbLoad()` method loads the bean as normal, including loading the version number from the database.
- The `ejbStore()` method checks the version number in the database versus its value when the EJB component was loaded.
    - If the version number has been modified, it means that there has been simultaneous access to

the EJB component and `ejbStore()` throws a `ConcurrentModificationException`.

- Otherwise, `ejbStore()` stores the data and completes as normal.
  The `ejbStore()` method performs this validation at the end of the transaction regardless of whether any data in the bean was modified.

Subsequent uses of the bean behave similarly, except that the `ejbLoad()` method loads its initial data (including the version number) from an internal cache. This saves a trip to the database. When the `ejbStore()` method is called, the version number is checked to ensure that the correct data was used in the transaction.

Version consistency is advantageous when you have EJB components that are rarely modified, because it allows two transactions to use the same EJB component at the same time. Because neither transaction modifies the data, the version number is unchanged at the end of both transactions, and both succeed. But now the transactions can run in parallel. If two transactions occasionally modify the same EJB component, one will succeed and one will fail and can be retried using the new values—which can still be faster than serializing all access to the EJB component if the retries are infrequent enough (though now your application logic has to be prepared to perform the retry operation).

To use version consistency, the database schema for a particular table must include a column where the version can be stored. You then specify that table in the `sun-cmp-mapping.xml` deployment descriptor for a particular bean:

```
<entity-mapping>
    <cmp-field-mapping>
        ...
    </cmp-field-mapping>
    <consistency>
        <check-version-of-accessed-instances>
            <column-name>OrderTable.VC_VERSION_NUMBER</column-name>
        </check-version-of-accessed-instances>
    </consistency>
</entity-mapping>
```

In addition, you must establish a trigger on the database to automatically update the version column when data in the specified table is modified. The GlassFish Server requires such a trigger to use version consistency. Having such a trigger also ensures that external applications that modify the EJB data will not conflict with EJB transactions in progress.

For example, the following DDL illustrates how to create a trigger for the `Order` table:

```
CREATE TRIGGER OrderTrigger
  BEFORE UPDATE ON OrderTable
  FOR EACH ROW
  WHEN (new.VC_VERSION_NUMBER = old.VC_VERSION_NUMBER)
  DECLARE
  BEGIN
    :NEW.VC_VERSION_NUMBER := :OLD.VC_VERSION_NUMBER + 1;
  END;
```

Request Partitioning

Request partitioning enables you to assign a request priority to an EJB component. This gives you the flexibility to make certain EJB components execute with higher priorities than others.

An EJB component which has a request priority assigned to it will have its requests (services) executed within an assigned threadpool. By assigning a threadpool to its execution, the EJB component can execute independently of other pending requests. In short, request partitioning enables you to meet service-level agreements that have differing levels of priority assigned to different services.

Request partitioning applies only to remote EJB components (those that implement a remote interface). Local EJB components are executed in their calling thread (for example, when a servlet calls a local bean, the local bean invocation occurs on the servlet's thread).

To Enable Request Partitioning

Follow this procedure.

1. Configure additional threadpools for EJB execution.
   Using the GlassFish Server Administration Console, navigate to the Configurations>configuration-name>Thread Pools node. Refer to the Administration Console online help for more information. Alternatively, you can follow the instructions in "Administering Thread Pools" in GlassFish Server Open Source Edition Administration Guide.
   Configure the threadpools as follows:
2. Add the additional threadpool IDs to the GlassFish Server's ORB.
   This can be done on the Configurations>configuration-name>ORB node in the Administration Console.
   For example, enable threadpools named `priority-1` and `priority-2` to the `<orb>` element as follows:

```
<orb max-connections="1024" message-fragment-size="1024"
    use-thread-pool-ids="thread-pool-1,priority-1,priority-2">
```

1. Include the threadpool ID in the `use-thread-pool-id` element of the EJB component's `sun-ejb-jar.xml` deployment descriptor.
   For example, the following `sun-ejb-jar.xml` deployment descriptor for an EJB component named "TheGreeter" is assigned to a thread pool named `priority-2`:

```
<sun-ejb-jar>
  <enterprise-beans>
    <unique-id>1</unique-id>
    <ejb>
      <ejb-name>TheGreeter</ejb-name>
      <jndi-name>greeter</jndi-name>
      <use-thread-pool-id>priority-1</use-thread-pool-id>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

1. Restart the GlassFish Server.

Tuning Tips for Specific Types of EJB Components

This section provides tips for tuning various specific types of EJB components:

- Entity Beans
- Stateful Session Beans
- Stateless Session Beans
- Read-Only Entity Beans
- Pre-Fetching Container Managed Relationship (CMR) Beans

These components can all be configured in the GlassFish Server Administration Console from the Configurations>configuration-name>EJB Container node. Alternatively, you can perform these configurations by following the instructions in "RMI-IIOP Load Balancing and Failover" in GlassFish Server Open Source Edition High Availability Administration Guide.

Entity Beans

Depending on the usage of a particular entity bean, one should tune `max-cache-size` so that the beans that are used less frequently (for example, an order that is created and never used after the transaction is over) are cached less, and beans that are used frequently (for example, an item in the inventory that gets referenced very often), are cached more.

Stateful Session Beans

When a stateful bean represents a user, a reasonable `max-cache-size` of beans is the expected number of concurrent users on the application server process. If this value is too low (in relation to the steady load of users), beans would be frequently passivated and activated, causing a negative impact on the response times, due to CPU intensive serialization and deserialization as well as disk I/O.

Another important variable for tuning is `cache-idle-timeout-in-seconds` where at periodic intervals of `cache-idle-timeout-in-seconds`, all the beans in the cache that have not been accessed for more than `cache-idle-timeout-in-seconds` time, are passivated. Similar to an HTTP session timeout, the bean is removed after it has not been accessed for `removal-timeout-in-seconds`. Passivated beans are stored on disk in serialized form. A large number of passivated beans could not only mean many files on the disk system, but also slower response time as the session state has to be de-serialized before the invocation.

Checkpoint only when needed

In high availability mode, when using stateful session beans, consider checkpointing only those methods that alter the state of the bean significantly. This reduces the number of times the bean state has to be checkpointed into the persistent store.

Stateless Session Beans

Stateless session beans are more readily pooled than entity or the stateful session beans. Valid values for `steady-pool-size`, `pool-resize-quantity` and `max-pool-size` are the best tunables for these type of beans. Set the `steady-pool-size` to greater than zero if you want to pre-populate the pool. This way, when the container comes up, it creates a pool with `steady-pool-size` number of beans. By pre-populating the pool it is possible to avoid the object creation time during method invocations.

Setting the `steady-pool size` to a very large value can cause unwanted memory growth and can result in large garbage collection times. `pool-resize-quantity` determines the rate of growth as well as the rate of decay of the pool. Setting it to a small value is better as the decay behaves like an exponential decay. Setting a small `max-pool-size` can cause excessive object destruction (and as a result excessive object creation) as instances are destroyed from the pool if the current pool size exceeds `max-pool-size`.

Read-Only Entity Beans

Read-only entity beans cache data from the database. GlassFish Server supports read-only beans that use both bean-managed persistence (BMP) and container-managed persistence (CMP). Of the two types, CMP read-only beans provide significantly better performance. In the EJB lifecycle, the EJB container

calls the `ejbLoad()` method of a read-only bean once. The container makes multiple copies of the EJB component from that data, and since the beans do not update the database, the container never calls the `ejbStore()` method. This greatly reduces database traffic for these beans.

If there is a bean that never updates the database, use a read-only bean in its place to improve performance. A read-only bean is appropriate if either:

- Database rows represented by the bean do not change.
- The application can tolerate using out-of-date values for the bean.

For example, an application might use a read-only bean to represent a list of best-seller books. Although the list might change occasionally in the database (say, from another bean entirely), the change need not be reflected immediately in an application.

The `ejbLoad()` method of a read-only bean is handled differently for CMP and BMP beans. For CMP beans, the EJB container calls `ejbLoad()` only once to load the data from the database; subsequent uses of the bean just copy that data. For BMP beans, the EJB container calls `ejbLoad()` the first time a bean is used in a transaction. Subsequent uses of that bean within the transaction use the same values. The container calls `ejbLoad()` for a BMP bean that doesn't run within a transaction every time the bean is used. Therefore, read-only BMP beans still make a number of calls to the database.

To create a read-only bean, add the following to the EJB deployment descriptor `sun-ejb-jar.xml`:

```
<is-read-only-bean>true</is-read-only-bean>
<refresh-period-in-seconds>600</refresh-period-in-seconds>
```

Refresh Period

An important parameter for tuning read-only beans is the refresh period, represented by the deployment descriptor entity `refresh-period-in-seconds`. For CMP beans, the first access to a bean loads the bean's state. The first access after the refresh period reloads the data from the database. All subsequent uses of the bean uses the newly refreshed data (until another refresh period elapses). For BMP beans, an `ejbLoad()` method within an existing transaction uses the cached data unless the refresh period has expired (in which case, the container calls `ejbLoad()` again).

This parameter enables the EJB component to periodically refresh its "snapshot" of the database values it represents. If the refresh period is less than or equal to 0, the bean is never refreshed from the database (the default behavior if no refresh period is given).

Pre-Fetching Container Managed Relationship (CMR) Beans

If a container-managed relationship (CMR) exists in your application, loading one bean will load all its

related beans. The canonical example of CMR is an order-orderline relationship where you have one `Order` EJB component that has related `OrderLine` EJB components. In previous releases of the application server, to use all those beans would require multiple database queries: one for the `Order` bean and one for each of the `OrderLine` beans in the relationship.

In general, if a bean has n relationships, using all the data of the bean would require n+1 database accesses. Use CMR pre-fetching to retrieve all the data for the bean and all its related beans in one database access.

For example, you have this relationship defined in the `ejb-jar.xml` file:

```
<relationships>
    <ejb-relation>
        <description>Order-OrderLine</description>
        <ejb-relation-name>Order-OrderLine</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Order-has-N-OrderLines
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>OrderEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>orderLines</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

When a particular `Order` is loaded, you can load its related `OrderLines` by adding this to the `sun-cmp-mapping.xml` file for the application:

```
<entity-mapping>
    <ejb-name>Order</ejb-name>
    <table-name>...</table-name>
    <cmp-field-mapping>...</cmp-field-mapping>
    <cmr-field-mapping>
        <cmr-field-name>orderLines</cmr-field-name>
        <column-pair>
            <column-name>OrderTable.OrderID</column-name>
            <column-name>OrderLineTable.OrderLine_OrderID</column-name>
        </column-pair>
        <fetched-with>
            <default>
        </fetched-with>
    </cmr-field-mapping>
</entity-mappping>
```

Now when an `Order` is retrieved, the CMP engine issues SQL to retrieve all related `OrderLines` with a `SELECT` statement that has the following `WHERE` clause:

```
OrderTable.OrderID = OrderLineTable.OrderLine_OrderID
```

This clause indicates an outer join. These `OrderLines` are pre-fetched.

Pre-fetching generally improves performance because it reduces the number of database accesses. However, if the business logic often uses `Orders` without referencing their `OrderLines`, then this can have a performance penalty, that is, the system has spent the effort to pre-fetch the `OrderLines` that are not actually needed.

Avoid pre-fetching for specific finder methods; this can often avoid that penalty. For example, consider an order bean has two finder methods: a `findByPrimaryKey` method that uses the `Orderlines`, and a `findByCustomerId` method that returns only order information and therefore does not use the `Orderlines`. If you have enabled CMR pre-fetching for the `Orderlines`, both finder methods will pre-fetch the `Orderlines`. However, you can prevent pre-fetching for the `findByCustomerId` method by including this information in the `sun-ejb-jar.xml` descriptor:

```
<ejb>
    <ejb-name>OrderBean</ejb-name>
    ...
    <cmp>
        <prefetch-disabled>
            <query-method>
                <method-name>findByCustomerId</method-name>
            </query-method>
        </prefetch-disabled>
     </cmp>
</ejb>
```

JDBC and Database Access

The following are some tips to improve the performance of database access:

- Use JDBC Directly
- Encapsulate Business Logic in Entity EJB Components
- Close Connections
- Minimize the Database Transaction Isolation Level

Use JDBC Directly

When dealing with large amounts of data, such as searching a large database, use JDBC directly rather than using Entity EJB components.

Encapsulate Business Logic in Entity EJB Components

Combine business logic with the Entity EJB component that holds the data needed for that logic to process.

Close Connections

To ensure that connections are returned to the pool, always close the connections after use.

Minimize the Database Transaction Isolation Level

Use the default isolation level provided by the JDBC driver rather than calling `setTransactionIsolationLevel()`, unless you are certain that your application behaves correctly and performs better at a different isolation level.

Reduce the database transaction isolation level when appropriate. Reduced isolation levels reduce work in the database tier, and could lead to better application performance. However, this must be done after carefully analyzing the database table usage patterns.

To set the database transaction isolation level using the GlassFish Server Administration Console, navigate to the Resources>JDBC>JDBC Connection Pools>pool-name node. Refer to the Administration Console online help for complete instructions. Alternatively, follow the instructions in "Administering Database Connectivity" in GlassFish Server Open Source Edition Administration Guide. For more information on tuning JDBC connection pools, see JDBC Connection Pool Settings.

Tuning Message-Driven Beans

This section provides some tips to improve performance when using JMS with message-driven beans (MDBs).

Use `getConnection()`

JMS connections are served from a connection pool. This means that calling `getConnection()` on a Queue connection factory is fast.

Tune the Message-Driven Bean's Pool Size

The container for message-driven beans (MDB) is different than the containers for entity and session beans. In the MDB container, sessions and threads are attached to the beans in the MDB pool. This design makes it possible to pool the threads for executing message-driven requests in the container.

Tune the Message-Driven bean's pool size to optimize the concurrent processing of messages. Set the size of the MDB pool to, based on all the parameters of the server (taking other applications into account). For example, a value greater than 500 is generally too large.

To configure MDB pool settings in the GlassFish Server Administration Console, navigate to the Configurations>configuration-name>EJB Container node and then select the MDB Settings tab. Refer to the Administration Console online help for more information. Alternatively, you can set the MDB pool size by using the following `asadmin set` subcommand:

```
asadmin set server.mdb-container.max-pool-size = value
```

Cache Bean-Specific Resources

Use the `setMessageDrivenContext()` or `ejbCreate()` method to cache bean specific resources, and release those resources from the `ejbRemove()` method.

Limit Use of JMS Connections

When designing an application that uses JMS connections make sure you use a methodology that sparingly uses connections, by either pooling them or using the same connection for multiple sessions.

The JMS connection uses two threads and the sessions use one thread each. Since these threads are not taken from a pool and the resultant objects aren't pooled, you could run out of memory during periods of heavy usage.

One workaround is to move `createTopicConnection` into the `init` of the servlet.

Make sure to specifically close the session, or it will stay open, which ties up resources.

3 Tuning the GlassFish Server

This chapter describes some ways to tune your GlassFish Server installation for optimum performance.

Note that while this chapter describes numerous interactions with both the GlassFish Server Administration Console and the command-line interface, it is not intended to provide exhaustive descriptions of either. For complete information about using the Administration Console, refer to the Administration Console online help. For complete information about using the GlassFish Server command-line interface, refer to the other titles in the GlassFish Server documentation set at `http://docs.oracle.com/docs/cd/E18930_01/index.html`.

The following topics are addressed here:

Using the GlassFish Server Performance Tuner

You can significantly improve the performance of GlassFish Server and the applications deployed on it by adjusting a few deployment and server configuration settings. These changes can be made manually, as described in this chapter, or by using the built-in Performance Tuner in the GlassFish Server Administration Console.

The Performance Tuner recommends server settings to suit the needs of your GlassFish Server deployment. It helps you reach an optimal configuration, although finer tuning might be needed in case of specific requirements. You can configure performance tuning for the entire domain, or for

individual GlassFish Server instances or clusters. The Tuner performs a static analysis of GlassFish Server resources and throughput requirements. Note that no dynamic inspection of the system is performed.

For complete information about using the Performance Tuning features available through the GlassFish Server Administration Console, refer to the Administration Console online help. You may also want to refer to the following resources for additional information:

- To view a demonstration video showing how to use the GlassFish Server Performance Tuner, see the Oracle GlassFish Server 3.1 - Performance Tuner demo (http://www.youtube.com/watch?v=FavsE2pzAjc).
- To find additional Performance Tuning development information, see the Performance Tuner in Oracle GlassFish Server 3.1 (http://blogs.oracle.com/jenblog/entry/performance_tuner_in_oracle_glassfish) blog.

Deployment Settings

Deployment settings can have significant impact on performance. Follow these guidelines when configuring deployment settings for best performance:

- Disable Auto-Deployment
- Use Pre-compiled JavaServer Pages
- Disable Dynamic Application Reloading

Disable Auto-Deployment

Enabling auto-deployment will adversely affect deployment, though it is a convenience in a development environment. For a production system, disable auto-deploy to optimize performance. If auto-deployment is enabled, then the Reload Poll Interval setting can have a significant performance impact.

To enable or disable auto-deployment from the GlassFish Server Administration Console, navigate to the Domain node and then click the Applications Configuration tab. Refer to the Administration Console for further instructions. Alternatively, refer to "To Deploy an Application or Module Automatically" in GlassFish Server Open Source Edition Application Deployment Guide for instructions on enabling or disabling auto-deployment.

Use Pre-compiled JavaServer Pages

Compiling JSP files is resource intensive and time consuming. Pre-compiling JSP files before deploying applications on the server will improve application performance. When you do so, only the resulting servlet class files will be deployed.

You can specify to precompile JSP files when you deploy an application through the Administration Console or `deploy` subcommand. You can also specify to pre-compile JSP files for a deployed application with the Administration Console. Navigate to the Domain node and then click the Applications Configuration tab. Refer to the Administration Console for further instructions.

Disable Dynamic Application Reloading

If dynamic reloading is enabled, the server periodically checks for changes in deployed applications and automatically reloads the application with the changes. Dynamic reloading is intended for development environments and is also incompatible with session persistence. To improve performance, disable dynamic class reloading.

You can use the Administration Console to disable dynamic class reloading for an application that is already deployed. Navigate to the Domain node and then click the Applications Configuration tab. Refer to the Administration Console for further instructions.

Logger Settings

The GlassFish Server produces writes log messages and exception stack trace output to the log file in the logs directory of the instance, domain-dir`/logs`. The volume of log activity can impact server performance; particularly in benchmarking situations.

General Settings

In general, writing to the system log slows down performance slightly; and increased disk access (increasing the log level, decreasing the file rotation limit or time limit) also slows down the application.

Also, make sure that any custom log handler does not log to a slow device like a network file system since this can adversely affect performance.

Log Levels

Set the log level for the server and its subsystems in the GlassFish Server Administration Console. Navigate to the Configurations>configuration-name>Logger Settings page, and follow the instructions in the online help. Alternatively, you can configure logging by following the instructions in "Administering the Logging Service" in GlassFish Server Open Source Edition Administration Guide.

Web Container Settings

Set Web container settings in the GlassFish Server Administration Console by navigating to the Configurations>configuration-name>Web Container node. Follow the instructions in the online help for more information. Alternatively, you can configure Web container settings by following the instructions in "Administering Web Applications" in GlassFish Server Open Source Edition Administration Guide.

- Session Properties: Session Timeout
- Manager Properties: Reap Interval
- Disable Dynamic JSP Reloading

Session Properties: Session Timeout

Session timeout determines how long the server maintains a session if a user does not explicitly invalidate the session. The default value is 30 minutes. Tune this value according to your application requirements. Setting a very large value for session timeout can degrade performance by causing the server to maintain too many sessions in the session store. However, setting a very small value can cause the server to reclaim sessions too soon.

Manager Properties: Reap Interval

Modifying the reap interval can improve performance, but setting it without considering the nature of your sessions and business logic can cause data inconsistency, especially for time-based persistence-frequency.

For example, if you set the reap interval to 60 seconds, the value of session data will be recorded every 60 seconds. But if a client accesses a servlet to update a value at 20 second increments, then inconsistencies will result.

For example, consider the following online auction scenario:

- Bidding starts at $5, in 60 seconds the value recorded will be $8 (three 20 second intervals).
- During the next 40 seconds, the client starts incrementing the price. The value the client sees is $10.
- During the client's 20 second rest, the GlassFish Server stops and starts in 10 seconds. As a result, the latest value recorded at the 60 second interval ($8) is be loaded into the session.
- The client clicks again expecting to see $11; but instead sees is $9, which is incorrect.
- So, to avoid data inconsistencies, take into the account the expected behavior of the application when adjusting the reap interval.

Disable Dynamic JSP Reloading

On a production system, improve web container performance by disabling dynamic JSP reloading. To do so, edit the `default-web.xml` file in the `config` directory for each instance. Change the servlet definition for a JSP file to look like this:

```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  <init-param>
    <param-name>development</param-name>
    <param-value>false</param-value>
  </init-param>
  <init-param>
    <param-name>xpoweredBy</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>genStrAsCharArray</param-name>
    <param-value>true</param-value>
  </init-param>  <load-on-startup>3</load-on-startup>
</servlet>
```

EJB Container Settings

The EJB Container has many settings that affect performance. As with other areas, use monitor the EJB Container to track its execution and performance.

You can configure most EJB container settings from the GlassFish Server Administration Console by navigating to the Configurations>configuration-name>EJB Container node and then following the instructions in the online help.

Monitoring the EJB Container

Monitoring the EJB container is disabled by default. You can enable EJB monitoring through the GlassFish Server Administration Console by navagating to the the Configurations>configuration-name>Monitoring node and then following the instructions in the online help. Set the monitoring level to LOW for to monitor all deployed EJB components, EJB pools, and EJB caches. Set the monitoring level to HIGH to also monitor EJB business methods.

Tuning the EJB Container

The EJB container caches and pools EJB components for better performance. Tuning the cache and pool properties can provide significant performance benefits to the EJB container.

The pool settings are valid for stateless session and entity beans while the cache settings are valid for stateful session and entity beans.

The following topics are addressed here:

- Overview of EJB Pooling and Caching
- Tuning the EJB Pool
- Tuning the EJB Cache
- Pool and Cache Settings for Individual EJB Components
- Commit Option

Overview of EJB Pooling and Caching

Both stateless session beans and entity beans can be pooled to improve server performance. In addition, both stateful session beans and entity beans can be cached to improve performance.

Table 3-1 Bean Type Pooling or Caching

Bean Type	Pooled	Cached
Stateless Session	Yes	No
Stateful Session	No	Yes
Entity	Yes	Yes

The difference between a pooled bean and a cached bean is that pooled beans are all equivalent and

indistinguishable from one another. Cached beans, on the contrary, contain conversational state in the case of stateful session beans, and are associated with a primary key in the case of entity beans. Entity beans are removed from the pool and added to the cache on `ejbActivate()` and removed from the cache and added to the pool on `ejbPassivate()`. `ejbActivate()` is called by the container when a needed entity bean is not in the cache. `ejbPassivate()` is called by the container when the cache grows beyond its configured limits.

Note:

If you develop and deploy your EJB components using Oracle Java Studio, then you need to edit the individual bean descriptor settings for bean pool and bean cache. These settings might not be suitable for production-level deployment.

Tuning the EJB Pool

A bean in the pool represents the pooled state in the EJB lifecycle. This means that the bean does not have an identity. The advantage of having beans in the pool is that the time to create a bean can be saved for a request. The container has mechanisms that create pool objects in the background, to save the time of bean creation on the request path.

Stateless session beans and entity beans use the EJB pool. Keeping in mind how you use stateless session beans and the amount of traffic your server handles, tune the pool size to prevent excessive creation and deletion of beans.

EJB Pool Settings

An individual EJB component can specify cache settings that override those of the EJB container in the `<bean-pool>` element of the EJB component's `sun-ejb-jar.xml` deployment descriptor.

The EJB pool settings are:

- Initial and Minimum Pool Size: the initial and minimum number of beans maintained in the pool. Valid values are from 0 to `MAX_INTEGER`, and the default value is 8. The corresponding EJB deployment descriptor attribute is `steady-pool-size`.
  Set this property to a number greater than zero for a moderately loaded system. Having a value greater than zero ensures that there is always a pooled instance to process an incoming request.
- Maximum Pool Size: the maximum number of connections that can be created to satisfy client requests. Valid values are from zero to `MAX_INTEGER`., and the default is 32. A value of zero means that the size of the pool is unbounded. The potential implication is that the JVM heap will be filled with objects in the pool. The corresponding EJB deployment descriptor attribute is `max-pool-size`.
  Set this property to be representative of the anticipated high load of the system. An very large pool

wastes memory and can slow down the system. A very small pool is also inefficient due to contention.

- Pool Resize Quantity: the number of beans to be created or deleted when the cache is being serviced by the server. Valid values are from zero to `MAX_INTEGER` and default is 16. The corresponding EJB deployment descriptor attribute is `resize-quantity`.
  Be sure to re-calibrate the pool resize quantity when you change the maximum pool size, to maintain an equilibrium. Generally, a larger maximum pool size should have a larger pool resize quantity.
- Pool Idle Timeout: the maximum time that a stateless session bean, entity bean, or message-driven bean is allowed to be idle in the pool. After this time, the bean is destroyed if the bean in case is a stateless session bean or a message driver bean. This is a hint to server. The default value is 600 seconds. The corresponding EJB deployment descriptor attribute is `pool-idle-timeout-in-seconds`.
  If there are more beans in the pool than the maximum pool size, the pool drains back to initial and minimum pool size, in steps of pool resize quantity at an interval specified by the pool idle timeout. If the resize quantity is too small and the idle timeout large, you will not see the pool draining back to steady size quickly enough.

Tuning the EJB Cache

A bean in the cache represents the ready state in the EJB lifecycle. This means that the bean has an identity (for example, a primary key or session ID) associated with it.

Beans moving out of the cache have to be passivated or destroyed according to the EJB lifecycle. Once passivated, a bean has to be activated to come back into the cache. Entity beans are generally stored in databases and use some form of query language semantics to load and store data. Session beans have to be serialized when storing them upon passivation onto the disk or a database; and similarly have to be deserialized upon activation.

Any incoming request using these "ready" beans from the cache avoids the overhead of creation, setting identity, and potentially activation. So, theoretically, it is good to cache as many beans as possible. However, there are drawbacks to caching:

- Memory consumed by all the beans affects the heap available in the Virtual Machine.
- Increasing objects and memory taken by cache means longer, and possibly more frequent, garbage collection.
- The application server might run out of memory unless the heap is carefully tuned for peak loads.

Keeping in mind how your application uses stateful session beans and entity beans, and the amount of traffic your server handles, tune the EJB cache size and timeout settings to minimize the number of activations and passivations.

EJB Cache Settings

An individual EJB component can specify cache settings that override those of the EJB container in the `<bean-cache>` element of the EJB component's `sun-ejb-jar.xml` deployment descriptor.

The EJB cache settings are:

- Max Cache Size: Maximum number of beans in the cache. Make this setting greater than one. The default value is 512. A value of zero indicates the cache is unbounded, which means the size of the cache is governed by Cache Idle Timeout and Cache Resize Quantity. The corresponding EJB deployment descriptor attribute is `max-cache-size`.
- Cache Resize Quantity: Number of beans to be created or deleted when the cache is serviced by the server. Valid values are from zero to MAX_INTEGER, and the default is 16. The corresponding EJB deployment descriptor attribute is `resize-quantity`.
- Removal Timeout: Amount of time that a stateful session bean remains passivated (idle in the backup store). If a bean was not accessed after this interval of time, then it is removed from the backup store and will not be accessible to the client. The default value is 60 minutes. The corresponding EJB deployment descriptor attribute is `removal-timeout-in-seconds`.
- Removal Selection Policy: Algorithm used to remove objects from the cache. The corresponding EJB deployment descriptor attribute is `victim-selection-policy`. Choices are:
  - NRU (not recently used). This is the default, and is actually pseudo-random selection policy.
  - FIFO (first in, first out)
  - LRU (least recently used)
- Cache Idle Timeout: Maximum time that a stateful session bean or entity bean is allowed to be idle in the cache. After this time, the bean is passivated to the backup store. The default value is 600 seconds. The corresponding EJB deployment descriptor attribute is `cache-idle-timeout-in-seconds`.
- Refresh period: Rate at which a read-only-bean is refreshed from the data source. Zero (0) means that the bean is never refreshed. The default is 600 seconds. The corresponding EJB deployment descriptor attribute is `refresh-period-in-seconds`. Note: this setting does not have a custom field in the Admin Console. To set it, use the Add Property button in the Additional Properties section.

Pool and Cache Settings for Individual EJB Components

Individual EJB pool and cache settings in the `sun-ejb-jar.xml` deployment descriptor override those of the EJB container. The following table lists the cache and pool settings for each type of EJB component.

Cache or Pool Setting	Stateful Session Bean	Stateless Session Bean	Entity Bean	Entity Read-Only Bean	Message Driven Bean
`cache-resize-quantity`	X	+	X	X	+

Cache or Pool Setting	Stateful Session Bean	Stateless Session Bean	Entity Bean	Entity Read-Only Bean	Message Driven Bean
max-cache-size	X	+	X	X	+
cache-idle-timeout-in-seconds	X	+	X	X	+
removal-timeout-in-seconds	X	+	X	X	+
victim-selection-policy	X	+	X	X	+
refresh-period-in-seconds	+	+	+	X	+
steady-pool-size	+	X	X	X	+
pool-resize-quantity	+	X	X	X	X
max-pool-size	+	X	X	X	X
pool-idle-timeout-in-seconds	+	X	X	X	X

Commit Option

The commit option controls the action taken by the EJB container when an EJB component completes a transaction. The commit option has a significant impact on performance.

The following are the possible values for the commit option:

- Commit option B: When a transaction completes, the bean is kept in the cache and retains its identity. The next invocation for the same primary key can use the cached instance. The EJB container will call the bean's `ejbLoad()` method before the method invocation to synchronize with the database.
- Commit option C: When a transaction completes, the EJB container calls the bean's `ejbPassivate()` method, the bean is disassociated from its primary key and returned to the free pool. The next invocation for the same primary key will have to get a free bean from the pool, set the `PrimaryKey` on this instance, and then call `ejbActivate()` on the instance. Again, the EJB container will call the bean's `ejbLoad()` before the method invocation to synchronize with the database.

Option B avoids `ejbAcivate()` and `ejbPassivate()` calls. So, in most cases it performs better than option C since it avoids some overhead in acquiring and releasing objects back to pool.

However, there are some cases where option C can provide better performance. If the beans in the cache are rarely reused and if beans are constantly added to the cache, then it makes no sense to cache beans. With option C is used, the container puts beans back into the pool (instead of caching them) after method invocation or on transaction completion. This option reuses instances better and reduces the number of live objects in the JVM, speeding garbage collection.

Determining the Best Commit Option

To determine whether to use commit option B or commit option C, first take a look at the cache-hits value using the monitoring command for the bean. If the cache hits are much higher than cache misses, then option B is an appropriate choice. You might still have to change the `max-cache-size` and `cache-resize-quantity` to get the best result.

If the cache hits are too low and cache misses are very high, then the application is not reusing the bean instances and hence increasing the cache size (using `max-cache-size`) will not help (assuming that the access pattern remains the same). In this case you might use commit option C. If there is no great difference between cache-hits and cache-misses then tune `max-cache-size`, and probably `cache-idle-timeout-in-seconds`.

Java Message Service Settings

The Type attribute that determines whether the Java Message Service (JMS) is on local or remote system affects performance. Local JMS performance is better than remote JMS performance. However, a remote cluster can provide failover capabilities and can be administrated together, so there may be other advantages of using remote JMS. For more information on using JMS, see "Administering the Java Message Service (JMS)" in GlassFish Server Open Source Edition Administration Guide.

Transaction Service Settings

The transaction manager makes it possible to commit and roll back distributed transactions.

A distributed transactional system writes transactional activity into transaction logs so that they can be recovered later. But writing transactional logs has some performance penalty.

The following topics are addressed here:

- Monitoring the Transaction Service
- Tuning the Transaction Service

Monitoring the Transaction Service

Transaction Manager monitoring is disabled by default. Enable monitoring of the transaction service through the GlassFish Server Administration Console by navigating to the Configurations>configuration-name>Monitoring node. Refer to the Administration Console for complete instructions.

You can also enable monitoring with these commands:

```
set serverInstance.transaction-service.monitoringEnabled=true
reconfig serverInstance
```

Viewing Monitoring Information

To view monitoring information for the transaction service in the GlassFish Server Administration Console, navigate to the server (Admin Server) node and then select the Monitor tab.

The following statistics are gathered on the transaction service:

- `total-tx-completed` Completed transactions.
- `total-tx-rolled-back` Total rolled back transactions.
- `total-tx-inflight` Total inflight (active) transactions.
- `isFrozen` Whether transaction system is frozen (true or false)
- `inflight-tx` List of inflight (active) transactions.

Tuning the Transaction Service

This property can be used to disable the transaction logging, where the performance is of utmost importance more than the recovery. This property, by default, won't exist in the server configuration.

Most Transaction Service tuning tasks can be performed through the GlassFish Server Administration Console by navigating to the Configurations>configuration-name>Transaction Service node and then following the instructions in the online help. Alternatively, you can follow the instructions in "Administering Transactions" in GlassFish Server Open Source Edition Administration Guide.

Disable Distributed Transaction Logging

You can disable transaction logging through the Administration Console or by using the following `asadmin set` subcommand:

```
asadmin set
server1.transaction-service.disable-distributed-transaction-logging=true
```

Disabling transaction logging can improve performance. Setting it to false (the default), makes the

transaction service write transactional activity to transaction logs so that transactions can be recovered. If Recover on Restart is checked, this property is ignored.

Set this property to true only if performance is more important than transaction recovery.

Recover On Restart (Automatic Recovery)

You can set the Recover on Restart attribute through the Administration Console or by entering the following `asadmin set` subcommand:

```
asadmin set server1.transaction-service.automatic-recovery=false
```

When Recover on Restart is true, the server will always perform transaction logging, regardless of the Disable Distributed Transaction Logging attribute.

If Recover on Restart is false, then:

- If Disable Distributed Transaction Logging is false (the default), then the server will write transaction logs.
- If Disable Distributed Transaction Logging is true, then the server will not write transaction logs. Not writing transaction logs will give approximately twenty percent improvement in performance, but at the cost of not being able to recover from any interrupted transactions. The performance benefit applies to transaction-intensive tests. Gains in real applications may be less.

Keypoint Interval

The keypoint interval determines how often entries for completed transactions are removed from the log file. Keypointing prevents a process log from growing indefinitely.

Frequent keypointing is detrimental to performance. The default value of the Keypoint Interval is 2048, which is sufficient in most cases.

HTTP Service Settings

Tuning the monitoring and access logging settings for the HTTP server instances that handle client requests are important parts of ensuring peak GlassFish Server performance.

The following topics are addressed here:

- Monitoring the HTTP Service

- HTTP Service Access Logging

Monitoring the HTTP Service

Disabling the collection of monitoring statistics can increase overall GlassFish Server performance. You can enable or disable monitoring statistics collection for the HTTP service using either the Administration Console or `asadmin` subcommands.

Refer to "Administering the Monitoring Service" in GlassFish Server Open Source Edition Administration Guide for complete instructions on configuring the monitoring service using `asadmin` subcommands.

If using the Administration Console, click the Configurations>configuration-name>Monitoring node for the configuration for which you want to enable or disable monitoring for selected components. Refer to the Administration Console online help for complete instructions.

For instructions on viewing comprehensive monitoring statistics using `asadmin` subcommands, see "Viewing Comprehensive Monitoring Data" in GlassFish Server Open Source Edition Administration Guide. If using the Administration Console, you can view monitoring statistics by navigating to the server (Admin Server) node, and then clicking the Monitor tab. Refer to the online help for configuring different views of the available monitoring statistics.

When viewing monitoring statistics, some key performance-related information to review includes the following:

- DNS Cache Information (dns)
- File Cache Information (file-cache)
- Keep Alive (keep-alive)
- Connection Queue

DNS Cache Information (dns)

The DNS cache caches IP addresses and DNS names. The DNS cache is disabled by default. In the DNS Statistics for Process ID All page under Monitor in the web-based Administration interface the following statistics are displayed:

- Enabled
- CacheEntries (CurrentCacheEntries / MaxCacheEntries)
- HitRatio
- Caching DNS Entries

- Limit DNS Lookups to Asynchronous
- Enabled
- NameLookups
- AddrLookups
- LookupsInProgress

Enabled

If the DNS cache is disabled, the rest of this section is not displayed.

By default, the DNS cache is off. Enable DNS caching in the Administration Console by clicking the Configurations>configuration-name>Network Config>http-listener-name node. Click the HTTP tab and enable the DNS Lookup option.

CacheEntries (CurrentCacheEntries / MaxCacheEntries)

The number of current cache entries and the maximum number of cache entries. A single cache entry represents a single IP address or DNS name lookup. Make the cache as large as the maximum number of clients that access your web site concurrently. Note that setting the cache size too high is a waste of memory and degrades performance.

Set the maximum size of the DNS cache by entering or changing the value in the in the Administration Console by clicking the Configurations>configuration-name>Network Config>http-listener-name node. Click the File tab and set the desired options.

HitRatio

The hit ratio is the number of cache hits divided by the number of cache lookups.

This setting is not tunable.

Note:

If you turn off DNS lookups on your server, host name restrictions will not work and IP addresses will appear instead of host names in log files.

Caching DNS Entries

It is possible to also specify whether to cache the DNS entries. If you enable the DNS cache, the server can store hostname information after receiving it. If the server needs information about the client in the future, the information is cached and available without further querying. specify the size of the DNS cache and an expiration time for DNS cache entries. The DNS cache can contain 32 to 32768 entries; the default value is 1024. Values for the time it takes for a cache entry to expire can range from 1 second to 1 year specified in seconds; the default value is 1200 seconds (20 minutes).

Limit DNS Lookups to Asynchronous

Do not use DNS lookups in server processes because they are resource-intensive. If you must include DNS lookups, make them asynchronous.

Enabled

If asynchronous DNS is disabled, the rest of this section will not be displayed.

NameLookups

The number of name lookups (DNS name to IP address) that have been done since the server was started. This setting is not tunable.

AddrLookups

The number of address loops (IP address to DNS name) that have been done since the server was started. This setting is not tunable.

LookupsInProgress

The current number of lookups in progress.

File Cache Information (file-cache)

The file cache caches static content so that the server handles requests for static content quickly. The file-cache section provides statistics on how your file cache is being used.

For information about tuning the file cache, see File Cache Settings.

The Monitoring page lists the following file cache statistics:

- Number of Hits on Cached File Content
- Number of Cache Entries
- Number of Hits on Cached File Info
- Heap Space Used for Cache
- Number of Misses on Cached File Content
- Cache Lookup Misses
- Number of Misses on Cached File Content
- Max Age of a Cache Entry
- Max Number of Cache Entries
- Max Number of Open Entries
- Is File Cached Enabled?
- Maximum Memory Map to be Used for Cache
- Memory Map Used for cache
- Cache Lookup Hits
- Open Cache Entries: The number of current cache entries and the maximum number of cache entries are both displayed. A single cache entry represents a single URI. This is a tunable setting.
- Maximum Heap Space to be Used for Cache

Keep Alive (keep-alive)

The following are statistics related to the Keep Alive system. The most important settings you can tune here relate to HTTP Timeout. See Timeout for more information.

- Connections Terminated Due to Client Connection Timed Out
- Max Connection Allowed in Keep-alive
- Number of Hits
- Connections in Keep-alive Mode
- Connections not Handed to Keep-alive Thread Due to too Many Persistent Connections
- The Time in Seconds Before Idle Connections are Closed
- Connections Closed Due to Max Keep-alive Being Exceeded

Connection Queue

- Total Connections Queued: Total connections queued is the total number of times a connection has been queued. This includes newly accepted connections and connections from the keep-alive system.
- Average Queuing Delay: Average queueing delay is the average amount of time a connection spends in the connection queue. This represents the delay between when a request connection is accepted by the server, and a request processing thread (also known as a session) begins servicing the request.

HTTP Service Access Logging

Accessing Logging can be tuned using several `asadmin` subcommands. Refer to "Administering the Monitoring Service" in GlassFish Server Open Source Edition Administration Guide for information about using these subcommands.

If using the Administration Console, Access Logging is configured from the Configurations>configuration-name>HTTP Service page. Refer to the Administration Console online help for complete instructions about the options on this page.

To enable or disable access logging, check or uncheck the Access Logging Enabled checkbox. Access Logging is disabled by default.

When performing benchmarking, ensure that Access Logging is disabled. If Access Logging is enabled, it is recommended that you also enable Rotation to ensure that the logs do not run out of disk space.

Network Listener Settings

You can tune Network Listener settings from the command line by using the instructions in "Administering HTTP Network Listeners" in GlassFish Server Open Source Edition Administration Guide.

If using the Administration Console, navigate to the Configurations >configuration-name>Network Config>Network Listeners>listener-name node, and then click the tab for the desired configuration page. Refer to the online help for complete instructions about the options on these tabs.

GlassFish Server Network Listener performance can be enhanced by modifying settings on the following Edit Network Listener tabs in the Administration Console:

- General Settings
- HTTP Settings

- File Cache Settings

General Settings

For machines with only one network interface card (NIC), set the network address to the IP address of the machine (for example, 192.18.80.23 instead of default 0.0.0.0). If you specify an IP address other than 0.0.0.0, the server will make one less system call per connection. Specify an IP address other than 0.0.0.0 for best possible performance. If the server has multiple NIC cards then create multiple listeners for each NIC.

HTTP Settings

The following settings on the HTTP tab can significantly affect performance:

- Max Connections
- DNS Lookup Enabled
- Timeout
- Header Buffer Length

Max Connections

Max Connections controls the number of requests that a particular client can make over a keep-alive connection. The range is any positive integer, and the default is 256.

Adjust this value based on the number of requests a typical client makes in your application. For best performance specify quite a large number, allowing clients to make many requests.

The number of connections specified by Max Connections is divided equally among the keep alive threads. If Max Connections is not equally divisible by Thread Count, the server can allow slightly more than Max Connections simultaneous keep alive connections.

DNS Lookup Enabled

This setting specifies whether the server performs DNS (domain name service) lookups on clients that access the server. When DNS lookup is not enabled, when a client connects, the server knows the client's IP address but not its host name (for example, it knows the client as 198.95.251.30, rather than `www.xyz.com`). When DNS lookup is enabled, the server will resolve the client's IP address into a host name for operations like access control, common gateway interface (CGI) programs, error reporting,

and access logging.

If the server responds to many requests per day, reduce the load on the DNS or NIS (Network Information System) server by disabling DNS lookup. Enabling DNS lookup will increase the latency and load on the system, so modify this setting with caution.

Timeout

Timeout determines the maximum time (in seconds) that the server holds open an HTTP keep alive connection. A client can keep a connection to the server open so that multiple requests to one server can be serviced by a single network connection. Since the number of open connections that the server can handle is limited, a high number of open connections will prevent new clients from connecting.

The default time out value is 30 seconds. Thus, by default, the server will close the connection if idle for more than 30 seconds. The maximum value for this parameter is 300 seconds (5 minutes).

The proper value for this parameter depends upon how much time is expected to elapse between requests from a given client. For example, if clients are expected to make requests frequently then, set the parameter to a high value; likewise, if clients are expected to make requests rarely, then set it to a low value.

Both HTTP 1.0 and HTTP 1.1 support the ability to send multiple requests across a single HTTP session. A server can receive hundreds of new HTTP requests per second. If every request was allowed to keep the connection open indefinitely, the server could become overloaded with connections. On Unix/Linux systems, this could easily lead to a file table overflow.

The GlassFish Server's Keep Alive system, which is affected by the Timeout setting, addresses this problem. A waiting keep alive connection has completed processing the previous request, and is waiting for a new request to arrive on the same connection. The server maintains a counter for the maximum number of waiting keep-alive connections. If the server has more than the maximum waiting connections open when a new connection waits for a keep-alive request, the server closes the oldest connection. This algorithm limits the number of open waiting keep-alive connections.

If your system has extra CPU cycles, incrementally increase the keep alive settings and monitor performance after each increase. When performance saturates (stops improving), then stop increasing the settings.

Header Buffer Length

The size (in bytes) of the buffer used by each of the request processing threads for reading the request data from the client.

Adjust the value based on the actual request size and observe the impact on performance. In most

cases the default should suffice. If the request size is large, increase this parameter.

File Cache Settings

The GlassFish Server uses a file cache to serve static information faster. The file cache contains information about static files such as HTML, CSS, image, or text files. Enabling the HTTP file cache will improve performance of applications that contain static files.

The following settings on the File Cache tab can significantly affect performance:

- Max File Count
- Max Age

Max File Count

Max File Count determines how many files are in the cache. If the value is too big, the server caches little-needed files, which wastes memory. If the value is too small, the benefit of caching is lost. Try different values of this attribute to find the optimal solution for specific applications—generally, the effects will not be great.

Max Age

This parameter controls how long cached information is used after a file has been cached. An entry older than the maximum age is replaced by a new entry for the same file.

If your Web site's content changes infrequently, increase this value for improved performance. Set the maximum age by entering or changing the value in the Maximum Age field of the File Cache Configuration page in the web-based Admin Console for the HTTP server node and selecting the File Caching Tab.

Set the maximum age based on whether the content is updated (existing files are modified) on a regular schedule or not. For example, if content is updated four times a day at regular intervals, you could set the maximum age to 21600 seconds (6 hours). Otherwise, consider setting the maximum age to the longest time you are willing to serve the previous version of a content file after the file has been modified.

Transport Settings

The Acceptor Threads property for the Transport service specifies how many threads you want in accept mode on a listen socket at any time. It is a good practice to set this to less than or equal to the number of CPUs in your system.

In the GlassFish Server, acceptor threads on an HTTP Listener accept connections and put them onto a connection queue. Session threads then pick up connections from the queue and service the requests. The server posts more session threads if required at the end of the request.

See "Administering HTTP Network Listeners" in GlassFish Server Open Source Edition Administration Guide for instructions on modifying the Acceptor Threads property. If using the Administration Console, the Acceptor Threads property is available on the Configurations>configuration-name>Network Config>Transports>tcp page.

Thread Pool Settings

You can tune thread pool settings by following the instructions in "Administering Thread Pools" in GlassFish Server Open Source Edition Administration Guide. If using the Administration Console Thread Pool settings are available on the Configurations>configuration-name>Thread Pools>thread-pool-name page.

The following thread pool settings can have significant effects on GlassFish Server performance:

- Max Thread Pool Size
- Min Thread Pool Size

Max Thread Pool Size

The Max Thread Pool Size parameter specifies the maximum number of simultaneous requests the server can handle. The default value is 5. When the server has reached the limit or request threads, it defers processing new requests until the number of active requests drops below the maximum amount. Increasing this value will reduce HTTP response latency times.

In practice, clients frequently connect to the server and then do not complete their requests. In these cases, the server waits a length of time specified by the Timeout parameter.

Also, some sites do heavyweight transactions that take minutes to complete. Both of these factors add to the maximum simultaneous requests that are required. If your site is processing many requests that take many seconds, you might need to increase the number of maximum simultaneous requests.

Adjust the thread count value based on your load and the length of time for an average request. In

general, increase this number if you have idle CPU time and requests that are pending; decrease it if the CPU becomes overloaded. If you have many HTTP 1.0 clients (or HTTP 1.1 clients that disconnect frequently), adjust the timeout value to reduce the time a connection is kept open.

Suitable Request Max Thread Pool Size values range from 100 to 500, depending on the load. If your system has extra CPU cycles, keep incrementally increasing thread count and monitor performance after each incremental increase. When performance saturates (stops improving), then stop increasing thread count.

Min Thread Pool Size

The Min Thread Pool Size property specifies the minimum number of threads the server initiates upon startup. The default value is 2. Min Thread Pool Size represents a hard limit for the maximum number of active threads that can run simultaneously, which can become a bottleneck for performance.

Specifying the same value for minimum and maximum threads allows GlassFish Server to use a slightly more optimized thread pool. This configuration should be considered unless the load on the server varies quite significantly.

ORB Settings

The GlassFish Server includes a high performance and scalable CORBA Object Request Broker (ORB). The ORB is the foundation of the EJB Container on the server.

For complete instructions on configuring ORB settings, refer to "Administering the Object Request Broker (ORB)" in GlassFish Server Open Source Edition Administration Guide. Also refer to "RMI-IIOP Load Balancing and Failover" in GlassFish Server Open Source Edition High Availability Administration Guide. You can also configure most ORB settings through the GlassFish Server Administration Console by navigating to the Configurations>configuration-name> ORB node and then following the instructions in the Administration Console online help.

The following topics are addressed here:

- Overview
- How a Client Connects to the ORB
- Monitoring the ORB
- Tuning the ORB

Overview

The ORB is primarily used by EJB components via:

- RMI/IIOP path from an application client (or rich client) using the application client container.
- RMI/IIOP path from another GlassFish Server instance ORB.
- RMI/IIOP path from another vendor's ORB.
- In-process path from the Web Container or MDB (message driven beans) container.

When a server instance makes a connection to another server instance ORB, the first instance acts as a client ORB. SSL over IIOP uses a fast optimized transport with high-performance native implementations of cryptography algorithms.

It is important to remember that EJB local interfaces do not use the ORB. Using a local interface passes all arguments by reference and does not require copying any objects.

How a Client Connects to the ORB

A rich client Java program performs a new `initialContext()` call which creates a client side ORB instance. This in turn creates a socket connection to the GlassFish Server IIOP port. The reader thread is started on the server ORB to service IIOP requests from this client. Using the `initialContext`, the client code does a lookup of an EJB deployed on the server. An IOR which is a remote reference to the deployed EJB on the server is returned to the client. Using this object reference, the client code invokes remote methods on the EJB.

`InitialContext` lookup for the bean and the method invocations translate the marshalling application request data in Java into IIOP message(s) that are sent on the socket connection that was created earlier on to the server ORB. The server then creates a response and sends it back on the same connection. This data in the response is then un-marshalled by the client ORB and given back to the client code for processing. The Client ORB shuts down and closes the connection when the rich client application exits.

Monitoring the ORB

ORB statistics are disabled by default. To gather ORB statistics, enable monitoring with the following `asadmin` command:

```
set serverInstance.iiop-service.orb.system.monitoringEnabled=true
reconfig serverInstance
```

If using the Administration Console, you can enable ORB monitoring on the Configurations>configuration-name>Monitoring page. To view ORB monitoring statistics through the Administration Console, navigate to the server (Admin Server) page and click the Monitor tab. Refer to the Administration Console online help for complete instructions.

The following statistics are of particular interest when tuning the ORB:

- Connection Statistics
- Thread Pools

Connection Statistics

The following statistics are gathered on ORB connections:

- `total-inbound-connections`: Total inbound connections to ORB.
- `total-outbound-connections`: Total outbound connections from ORB.

Use the following command to get ORB connection statistics:

```
asadmin get --monitor
    serverInstance.iiop-service.orb.system.orb-connection.*
```

Thread Pools

The following statistics are gathered on ORB thread pools:

- `thread-pool-size`: Number of threads in ORB thread pool.
- `waiting-thread-count`: Number of thread pool threads waiting for work to arrive.

Use the following command to display ORB thread pool statistics:

```
asadmin get --monitor
    serverInstance.iiop-service.orb.system.orb-thread-pool.*
```

Tuning the ORB

Tune ORB performance by setting ORB parameters and ORB thread pool parameters. You can often decrease response time by leveraging load-balancing, multiple shared connections, thread pool and

message fragment size. You can improve scalability by load balancing between multiple ORB servers from the client, and tuning the number of connection between the client and the server.

The following topics are addressed here:

- Tunable ORB Parameters
- ORB Thread Pool Parameters
- Client ORB Properties
- Thread Pool Sizing
- Examining IIOP Messages

Tunable ORB Parameters

You can tune ORB parameters using the instructions in "Administering the Object Request Broker (ORB)" in GlassFish Server Open Source Edition Administration Guide. If using the Administration Console, navigate to the Configurations>configuration-name>ORB node and refer to the online help.

The following table summarizes the tunable ORB parameters.

Table 3-2 Tunable ORB Parameters

Path	ORB Modules	Server Settings
RMI/ IIOP from application client to application server	communication infrastructure, thread pool	steady-thread-pool-size, max-thread-pool-size, idle-thread-timeout-in-seconds
RMI/ IIOP from ORB to GlassFish Server	communication infrastructure, thread pool	steady-thread-pool-size, max-thread-pool-size, idle-thread-timeout-in-seconds
RMI/ IIOP from a vendor ORB	parts of communication infrastructure, thread pool	steady-thread-pool-size, max-thread-pool-size, idle-thread-timeout-in-seconds
In-process	thread pool	steady-thread-pool-size, max-thread-pool-size, idle-thread-timeout-in-seconds

ORB Thread Pool Parameters

The ORB thread pool contains a task queue and a pool of threads. Tasks or jobs are inserted into the task queue and free threads pick tasks from this queue for execution. Do not set a thread pool size such

that the task queue is always empty. It is normal for a large application's Max Pool Size to be ten times the size of the current task queue.

The GlassFish Server uses the ORB thread pool to:

- Execute every ORB request
- Trim EJB pools and caches
- Execute MDB requests

Thus, even when one is not using ORB for remote-calls (via RMI/ IIOP), set the size of the threadpool to facilitate cleaning up the EJB pools and caches.

You can set ORB thread pool attributes using the instructions in "Administering Thread Pools" in GlassFish Server Open Source Edition Administration Guide. If using the Administration Console, navigate to Configurations>configuration-name> Thread Pools>thread-pool-name, where thread-pool-name is the thread pool ID selected for the ORB. Thread pools have the following attributes that affect performance.

- Minimum Pool Size: The minimum number of threads in the ORB thread pool. Set to the average number of threads needed at a steady (RMI/ IIOP) load.
- Maximum Pool Size: The maximum number of threads in the ORB thread pool.
- Idle Timeout: Number of seconds to wait before removing an idle thread from pool. Allows shrinking of the thread pool.
- Number of Work Queues

In particular, the maximum pool size is important to performance. For more information, see Thread Pool Sizing.

Client ORB Properties

Specify the following properties as command-line arguments when launching the client program. You do this by using the following syntax when starting the Java VM:

```
-Dproperty=value
```

The following topics are addressed here:

- Controlling Connections Between Client and Server ORB
- Limiting the Maximum Number of Client Connections
- Load Balancing

Controlling Connections Between Client and Server ORB

When using the default JDK ORB on the client, a connection is established from the client ORB to the application server ORB every time an initial context is created. To pool or share these connections when they are opened from the same process by adding to the configuration on the client ORB.

```
-Djava.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
```

Limiting the Maximum Number of Client Connections

You can specify the total maximum number of client connections on all ORB listener ports (TCP, SSL and SSL with mutual authentication). When open client connections exceed the maximum value you specify, the ORB rejects any new incoming client connections.

Set this value to support the expected number of simultaneous client connections, but not to exceed the VM or system file descriptor limits. If the value is set too high, the ORB will continue accepting new client connections, resulting in a "too many open files" error if the VM runs out of file descriptors.

To specify the maximum number of client connections, set the `configs.config.config-name.iiop-service.orb.max-connections` attribute to the number that you require:

```
asadmin> set configs.config.config-name.iiop-service.orb.max-connections=max-connections
```

config-name

The name of the configuration in which the IIOP service is defined. For example, `server-config` is the name for the configuration of the domain administration server (DAS).

max-connections

An integer that specifies the maximum number of client connections.

For updates to this value to take effect, restart GlassFish Server.

The following example shows how to set the maximum number of client connections for the ORB in the DAS to `512`:

```
asadmin> set configs.config.server-config.iiop-service.orb.max-connections=512
configs.config.server-config.iiop-service.orb.max-connections=512
Command set executed successfully.
```

Load Balancing

For information on how to configure HTTP load balancing, see "Configuring HTTP Load Balancing" in GlassFish Server Open Source Edition High Availability Administration Guide.

For information on how to configure RMI/IIOP for multiple application server instances in a cluster, "RMI-IIOP Load Balancing and Failover" in GlassFish Server Open Source Edition High Availability Administration Guide.

When tuning the client ORB for load-balancing and connections, consider the number of connections opened on the server ORB. Start from a low number of connections and then increase it to observe any performance benefits. A connection to the server translates to an ORB thread reading actively from the connection (these threads are not pooled, but exist currently for the lifetime of the connection).

Thread Pool Sizing

After examining the number of inbound and outbound connections as explained above, tune the size of the thread pool appropriately. This can affect performance and response times significantly.

The size computation takes into account the number of client requests to be processed concurrently, the resource (number of CPUs and amount of memory) available on the machine and the response times required for processing the client requests.

Setting the size to a very small value can affect the ability of the server to process requests concurrently, thus affecting the response times since requests will sit longer in the task queue. On the other hand, having a large number of worker threads to service requests can also be detrimental because they consume system resources, which increases concurrency. This can mean that threads take longer to acquire shared structures in the EJB container, thus affecting response times.

The worker thread pool is also used for the EJB container's housekeeping activity such as trimming the pools and caches. This activity needs to be accounted for also when determining the size. Having too many ORB worker threads is detrimental for performance since the server has to maintain all these threads. The idle threads are destroyed after the idle thread timeout period.

Examining IIOP Messages

It is sometimes useful to examine the IIOP messages passed by the GlassFish Server. To make the server save IIOP messages to the `server.log` file, set the JVM option `-Dcom.sun.CORBA.ORBDebug=giop`. Use the same option on the client ORB.

The following is an example of IIOP messages saved to the server log. Note: in the actual output, each line is preceded by the timestamp, such as `[29/Aug/2002:22:41:43] INFO (27179): CORE3282: stdout`.

```
 ++++++++++++++++++++++++++++++
 Message(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
createFromStream: type is 4 <
 MessageBase(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
Message GIOP version: 1.2
 MessageBase(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
ORB Max GIOP Version: 1.2
 Message(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
createFromStream: message construction complete.
 com.sun.corba.ee.internal.iiop.MessageMediator
(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]): Received message:
 ----- Input Buffer -----
 Current index: 0
 Total length : 340
 47 49 4f 50 01 02 00 04 0 0 00 01 48 00 00 00 05 GIOP.......H....
```

Note:

The flag `-Dcom.sun.CORBA.ORBdebug=giop` generates many debug messages in the logs. This is used only when you suspect message fragmentation.

In this sample output above, the `createFromStream` type is shown as `4`. This implies that the message is a fragment of a bigger message. To avoid fragmented messages, increase the fragment size. Larger fragments mean that messages are sent as one unit and not as fragments, saving the overhead of multiple messages and corresponding processing at the receiving end to piece the messages together.

If most messages being sent in the application are fragmented, increasing the fragment size is likely to improve efficiency. On the other hand, if only a few messages are fragmented, it might be more efficient to have a lower fragment size that requires smaller buffers for writing messages.

Resource Settings

Tuning JDBC and connector resources can significantly improve GlassFish Server performance.

The following topics are addressed here:

- JDBC Connection Pool Settings
- Connector Connection Pool Settings

JDBC Connection Pool Settings

For optimum performance of database-intensive applications, tune the JDBC Connection Pools managed by the GlassFish Server. These connection pools maintain numerous live database connections that can be reused to reduce the overhead of opening and closing database connections. This section describes how to tune JDBC Connection Pools to improve performance.

J2EE applications use JDBC Resources to obtain connections that are maintained by the JDBC Connection Pool. More than one JDBC Resource is allowed to refer to the same JDBC Connection Pool. In such a case, the physical connection pool is shared by all the resources.

Refer to "Administering Database Connectivity" in GlassFish Server Open Source Edition Administration Guide for more information about managing JDBC connection pools.

The following topics are addressed here:

- Monitoring JDBC Connection Pools
- Tuning JDBC Connection Pools

Monitoring JDBC Connection Pools

Statistics-gathering is disabled by default for JDBC Connection Pools. Refer to for instructions on enabling JDBC monitoring in "Administering the Monitoring Service" in GlassFish Server Open Source Edition Administration Guide. If using the Administration Console, JDBC monitoring can be enabled on the Configurations>configuration-name>Monitoring page.

The following attributes are monitored:

- `numConnFailedValidation (count)` Number of connections that failed validation.
- `numConnUsed (range)` Number of connections that have been used.
- `numConnFree (count)` Number of free connections in the pool.
- `numConnTimedOut (bounded range)` Number of connections in the pool that have timed out.

To get JDBC monitoring statistics, use the following commands:

```
asadmin get --monitor=true
serverInstance.resources.jdbc-connection-pool.*asadmin get
--monitor=true serverInstance.resources.jdbc-connection-pool. poolName.* *
```

To view JDBC monitoring statistics through the Administration Console, navigate to the server (Admin Server) page and click the Monitor tab. Refer to the Administration Console online help for complete instructions.

Tuning JDBC Connection Pools

Refer to "Administering Database Connectivity" in GlassFish Server Open Source Edition Administration Guide for instructions on configuring JDBC connection pools. If using the GlassFish Server Administration Console by navigating to the Resources>JDBC>JDBC Connection Pools>jdbc-pool-name page and then clicking the desired tab.

The following JDBC properites affect GlassFish Server performance:

- Pool Size Settings
- Timeout Settings
- Isolation Level Settings
- Connection Validation Settings

Pool Size Settings

Pool Size settings can be configured in the Pool Settings section on the General tab in the Edit JDBC Connection Pool page.

The following settings control the size of the connection pool:

- Initial and Mimimum Pool Size: Size of the pool when created, and its minimum allowable size.
- Maximum Pool Size: Upper limit of size of the pool.
- Pool Resize Quantity: Number of connections to be removed when the idle timeout expires. Connections that have idled for longer than the timeout are candidates for removal. When the pool size reaches the initial and minimum pool size, removal of connections stops.

The following table summarizes advantages and disadvantages to consider when sizing connection pools.

Table 3-3 Connection Pool Sizing

Connection Pool	Advantages	Disadvantages
Small Connection pool	Faster access on the connection table.	May not have enough connections to satisfy requests. Requests may spend more time in the queue.

Connection Pool	Advantages	Disadvantages
Large Connection pool	More connections to fulfill requests. Requests will spend less (or no) time in the queue	Slower access on the connection table.

Timeout Settings

The following JDBC timeout settings can be configured on the in the Pool Settings section on the General tab in the Edit JDBC Connection Pool page.

- Max Wait Time: Amount of time the caller (the code requesting a connection) will wait before getting a connection timeout. The default is 60 seconds. A value of zero forces caller to wait indefinitely.
  To improve performance set Max Wait Time to zero (0). This essentially blocks the caller thread until a connection becomes available. Also, this allows the server to alleviate the task of tracking the elapsed wait time for each request and increases performance.
- Idle Timeout: Maximum time in seconds that a connection can remain idle in the pool. After this time, the pool can close this connection. This property does not control connection timeouts on the database server.
  Keep this timeout shorter than the database server timeout (if such timeouts are configured on the database), to prevent accumulation of unusable connection in GlassFish Server.
  For best performance, set Idle Timeout to zero (0) seconds, so that idle connections will not be removed. This ensures that there is normally no penalty in creating new connections and disables the idle monitor thread. However, there is a risk that the database server will reset a connection that is unused for too long.

Isolation Level Settings

The following JDBC Isolation Level settings can be configured in the Transaction section on the General tab in the Edit JDBC Connection Pool page.

- Transaction Isolation: Specifies the transaction isolation level of the pooled database connections. If this parameter is unspecified, the pool uses the default isolation level provided by the JDBC Driver.
- Isolation Level Guaranteed: Guarantees that every connection obtained from the pool has the isolation specified for the Transaction Isolation level. Applicable only when the Transaction Isolation level is specified. The default value is Guaranteed.
  This setting can have some performance impact on some JDBC drivers. Set to false when certain that the application does not change the isolation level before returning the connection.

Avoid specifying the Transaction Isolation level. If that is not possible, consider disabling the Isolation

Level Guaranteed option and then make sure applications do not programmatically alter the connections; isolation level.

If you must specify a Transaction Isolation level, specify the best-performing level possible. The isolation levels listed from best performance to worst are:

1. `READ_UNCOMMITTED`
2. `READ_COMMITTED`
3. `REPEATABLE_READ`
4. `SERIALIZABLE`

Choose the isolation level that provides the best performance, yet still meets the concurrency and consistency needs of the application.

Connection Validation Settings

JDBC Connection Validation settings can be configured in the Connection Validation section on the Advanced tab in the Edit JDBC Connection Pool page.

- Connection Validation Required: If enabled, the pool validates connections (checks to find out if they are usable) before providing them to an application.
  If possible, keep this option disabled. Requiring connection validation forces the server to apply the validation algorithm every time the pool returns a connection, which adds overhead to the latency of `getConnection()`. If the database connectivity is reliable, you can omit validation.
- Validation Method: Specifies the type of connection validation to perform. Must be one of the following:
  - `auto-commit`: Attempt to perform an auto-commit on the connection.
  - `metadata`: Attempt to get metadata from the connection.
  - `table`: Performing the query on a specified table. If this option is selected, Table Name must also be set. Choosing this option may be necessary if the JDBC driver caches calls to `setAutoCommit()` and `getMetaData()`.
  - `custom-validation`: Define a custom validation method.
- Table Name: Name of the table to query when the Validation Method is set to `table`.
- Close All Connections On Any Failure: Specify whether all connections in the pool should be closed if a single validation check fails. This option is disabled by default. One attempt will be made to re-establish failed connections.

Connector Connection Pool Settings

From a performance standpoint, connector connection pools are similar to JDBC connection pools. Follow all the recommendations in the previous section, Tuning JDBC Connection Pools.

Transaction Support

You may be able to improve performance by overriding the default transaction support specified for each connector connection pool.

For example, consider a case where an Enterprise Information System (EIS) has a connection factory that supports local transactions with better performance than global transactions. If a resource from this EIS needs to be mixed with a resource coming from another resource manager, the default behavior forces the use of XA transactions, leading to lower performance. However, by changing the EIS's connector connection pool to use LocalTransaction transaction support and leveraging the Last Agent Optimization feature previously described, you could leverage the better-performing EIS LocalTransaction implementation. For more information on LAO, see Configure JDBC Resources as One-Phase Commit Resources.

You can specify transaction support when you create or edit a connector connection pool.

Also set transaction support using `asadmin`. For example, the following `asadmin` command could be used to create a connector connection pool `TESTPOOL` with `transaction-support` set to `LOCAL`.

```
asadmin> create-connector-connection-pool --raname jdbcra
--connectiondefinition javax.sql.DataSource
-transactionsupport LocalTransaction
TESTPOOL
```

Load Balancer Settings

GlassFish Server Open Source Edition is compatible with the Apache HTTP server `mod_jk` module for load balancing.

GlassFish Server load balancing configurations can vary widely depending on the needs of your enterprise and are beyond the scope of this Performance Tuning Guide. For complete information about configuring load balancing in GlassFish Server, refer to the following documentation:

- "Configuring HTTP Load Balancing" in GlassFish Server Open Source Edition High Availability Administration Guide
- "RMI-IIOP Load Balancing and Failover" in GlassFish Server Open Source Edition High Availability

Administration Guide

4 Tuning the Java Runtime System

The following topics are addressed here:

Java Virtual Machine Settings

Java SE 7 provides two implementations of the HotSpot Java virtual machine (JVM):

- The client VM is tuned for reducing startup time and memory footprint. Invoke it by using the `-client` JVM command-line option.
- The server VM is designed for maximum program execution speed. Invoke it by using the `-server` JVM command-line option.

By default, the GlassFish Server uses the JVM setting appropriate to the purpose:

- Developer Profile, targeted at application developers, uses the `-client` JVM flag to optimize startup performance and conserve memory resources.
- Enterprise Profile, targeted at production deployments, uses the `-server` JVM flag to maximize program execution speed.

You can override the default JVM options by following the instructions in "Administering JVM Options" in GlassFish Server Open Source Edition Administration Guide. If using the Administration Console, navigate to the Configurations>configuration-name>JVM Settings node, and then click the JVM Options tab. Refer to the online help for complete information about the settings on this page.

For more information on server-class machine detection in Java SE 7, see Server-Class Machine Detection (`http://docs.oracle.com/javase/7/docs/technotes/guides/vm/server-class.html`).

For more information on JVMs, see Java Virtual Machines (`http://docs.oracle.com/javase/7/docs/`).

Start Options

In some situations, performance can be improved by using large page sizes. For Ultrasparc CMT

systems, include the `-XX:+UseLargePages` and `-XX:LargePageSizeInBytes=256m` arguments with your JVM tuning.

Tuning High Availability Persistence

If session `s1` and `s2` need to be replicated to an instance (backup server), the replication module batches the replication messages to be sent to that instance instead of sending separate replication messages. This improves performance. In configurations in which a lot of session replication is performed, you may find better performance by tuning the `org.shoal.cache.transmitter.max.batch.size` system property. This property determines the number of replication messages that constitute one batch.

The default value for this property is `20`. You can try setting it as high as `90`, depending on system loads. Like all system properties, this property is set with the `-D` flag in your Java arguments.

Managing Memory and Garbage Collection

The efficiency of any application depends on how well memory and garbage collection are managed. The following sections provide information on optimizing memory and allocation functions:

- Tuning the Garbage Collector
- Tracing Garbage Collection
- Other Garbage Collector Settings
- Tuning the Java Heap
- Rebasing DLLs on Windows

Tuning the Garbage Collector

Garbage collection (GC) reclaims the heap space previously allocated to objects no longer needed. The process of locating and removing the dead objects can stall any application and consume as much as 25 percent throughput.

Almost all Java Runtime Environments come with a generational object memory system and sophisticated GC algorithms. A generational memory system divides the heap into a few carefully sized partitions called generations. The efficiency of a generational memory system is based on the observation that most of the objects are short lived. As these objects accumulate, a low memory condition occurs forcing GC to take place.

The heap space is divided into the old and the new generation. The new generation includes the new object space (eden), and two survivor spaces. The JVM allocates new objects in the eden space, and moves longer lived objects from the new generation to the old generation.

The young generation uses a fast copying garbage collector which employs two semi-spaces (survivor spaces) in the eden, copying surviving objects from one survivor space to the second. Objects that survive multiple young space collections are tenured, meaning they are copied to the tenured generation. The tenured generation is larger and fills up less quickly. So, it is garbage collected less frequently; and each collection takes longer than a young space only collection. Collecting the tenured space is also referred to as doing a full generation collection.

The frequent young space collections are quick (a few milliseconds), while the full generation collection takes a longer (tens of milliseconds to a few seconds, depending upon the heap size).

Other GC algorithms, such as the Concurrent Mark Sweep (CMS) algorithm, are incremental. They divide the full GC into several incremental pieces. This provides a high probability of small pauses. This process comes with an overhead and is not required for enterprise web applications.

When the new generation fills up, it triggers a minor collection in which the surviving objects are moved to the old generation. When the old generation fills up, it triggers a major collection which involves the entire object heap.

Both HotSpot and Solaris JDK use thread local object allocation pools for lock-free, fast, and scalable object allocation. So, custom object pooling is not often required. Consider pooling only if object construction cost is high and significantly affects execution profiles.

Choosing the Garbage Collection Algorithm

The default collector for Java server class machines will optimize for throughput but be tolerant of somewhat long pause times. If you would prefer to have minimal pause times at the expense of some throughput and increased CPU usage, consider using the CMS collector.

To use the CMS collector

Follow this procedure.

1. Make sure that the system is not using 100 percent of its CPU.
2. Configure the CMS collector in the server instance.
   To do this, add the following JVM options:
   - `-XX:+UseConcMarkSweepGC`
   - `-XX:SoftRefLRUPolicyMSPerMB=1`

Additional Information

Use the `jvmstat` utility to monitor HotSpot garbage collection. (See Further Information.)

For detailed information on tuning the garbage collector, see Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning (`http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`).

Tracing Garbage Collection

The two primary measures of garbage collection performance are throughput and pauses. Throughput is the percentage of the total time spent on other activities apart from GC. Pauses are times when an application appears unresponsive due to GC.

Two other considerations are footprint and promptness. Footprint is the working size of the JVM process, measured in pages and cache lines. Promptness is the time between when an object becomes dead, and when the memory becomes available. This is an important consideration for distributed systems.

A particular generation size makes a trade-off between these four metrics. For example, a large young generation likely maximizes throughput, but at the cost of footprint and promptness. Conversely, using a small young generation and incremental GC will minimize pauses, and thus increase promptness, but decrease throughput.

JVM diagnostic output will display information on pauses due to garbage collection. If you start the server in verbose mode (use the command `asadmin start-domain --verbose` domain), then the command line argument `-verbose:gc` prints information for every collection. Here is an example of output of the information generated with this JVM flag:

```
[GC 50650K->21808K(76868K), 0.0478645 secs]
 [GC 51197K->22305K(76868K), 0.0478645 secs]
 [GC 52293K->23867K(76868K), 0.0478645 secs]
 [Full GC 52970K->1690K(76868K), 0.54789968 secs]
```

On each line, the first number is the combined size of live objects before GC, the second number is the size of live objects after GC, the number in parenthesis is the total available space, which is the total heap minus one of the survivor spaces. The final figure is the amount of time that the GC took. This example shows three minor collections and one major collection. In the first GC, 50650 KB of objects existed before collection and 21808 KB of objects after collection. This means that 28842 KB of objects were dead and collected. The total heap size is 76868 KB. The collection process required 0.0478645 seconds.

Other useful monitoring options include:

- `-XX:+PrintGCDetails` for more detailed logging information

- `-Xloggc:file` to save the information in a log file

Other Garbage Collector Settings

To specify the attributes for the Java virtual machine, use the Administration Console and set the property under config-name > JVM settings (JVM options).

Setting the Maximum Permanent Generation

For applications that do not dynamically generate and load classes, the size of the permanent generation does not affect GC performance. For applications that dynamically generate and load classes (for example, JSP applications), the size of the permanent generation does affect GC performance, since filling the permanent generation can trigger a Full GC. Tune the maximum permanent generation with the `-XX:MaxPermSize` option.

Disabling Explicit Garbage Collection

Although applications can explicitly invoke GC with the `System.gc()` method, doing so is a bad idea since this forces major collections, and inhibits scalability on large systems. It is best to disable explicit GC by using the flag `-XX:+DisableExplicitGC`.

Note:

On Windows systems, setting the `-XX:+DisableExplicitGC` option might prevent the renaming or removal of open application files. As a result, deployment, redeployment, or other operations that attempt to rename or delete files might fail.

Application files can remain open because the files have been used by class loaders to find classes or resources, or have been opened explicitly by GlassFish Server or application code but never explicitly closed. On Windows systems, open files cannot be renamed or deleted. To overcome this limitation, GlassFish Server uses the `System.gc()` method to garbage collect the Java object that corresponds to an open file. When the Java object that corresponds to an open file is garbage collected, the object's `finalize` method closes the open channel to the file. GlassFish Server can then delete or rename the file.

Setting the Frequency of Full Garbage Collection

GlassFish Server uses RMI in the Administration module for monitoring. Garbage cannot be collected

in RMI-based distributed applications without occasional local collections, so RMI forces a periodic full collection. Control the frequency of these collections with the property `-sun.rmi.dgc.client.gcInterval`. For example, `- java -Dsun.rmi.dgc.client.gcInterval=3600000` specifies explicit collection once per hour instead of the default rate of once per minute.

Tuning the Java Heap

This section discusses topics related to tuning the Java Heap for performance.

- Guidelines for Java Heap Sizing
- Heap Tuning Parameters

Guidelines for Java Heap Sizing

Maximum heap size depends on maximum address space per process. The following table shows the maximum per-process address values for various platforms:

Table 4-1 Maximum Address Space Per Process

Operating System	Maximum Address Space Per Process
Oracle/Redhat/Ubuntu Linux 32-bit	4 GB
Oracle/Redhat/Ubuntu Linux 64-bit	Terabytes
Windows XP/2008/7	2 GB
Solaris x86 (32-bit)	4 GB
Solaris 32-bit	4 GB
Solaris 64-bit	Terabytes

Maximum heap space is always smaller than maximum address space per process, because the process also needs space for stack, libraries, and so on. To determine the maximum heap space that can be allocated, use a profiling tool to examine the way memory is used. Gauge the maximum stack space the process uses and the amount of memory taken up libraries and other memory structures. The difference between the maximum address space and the total of those values is the amount of memory that can be allocated to the heap.

You can improve performance by increasing your heap size or using a different garbage collector. In general, for long-running server applications, use the Java SE throughput collector on machines with multiple processors (`-XX:+AggressiveHeap`) and as large a heap as you can fit in the free memory of your machine.

Heap Tuning Parameters

You can control the heap size with the following JVM parameters:

- `-Xms`value
- `-Xmx`value
- `-XX:MinHeapFreeRatio=`minimum
- `-XX:MaxHeapFreeRatio=`maximum
- `-XX:NewRatio=`ratio
- `-XX:NewSize=`size
- `-XX:MaxNewSize=`size
- `-XX:+AggressiveHeap`

The `-Xms` and `-Xmx` parameters define the minimum and maximum heap sizes, respectively. Since GC occurs when the generations fill up, throughput is inversely proportional to the amount of the memory available. By default, the JVM grows or shrinks the heap at each GC to try to keep the proportion of free space to the living objects at each collection within a specific range. This range is set as a percentage by the parameters `-XX:MinHeapFreeRatio=`minimum and `-XX:MaxHeapFreeRatio=`maximum; and the total size bounded by `-Xms` and `-Xmx`.

Set the values of `-Xms` and `-Xmx` equal to each other for a fixed heap size. When the heap grows or shrinks, the JVM must recalculate the old and new generation sizes to maintain a predefined `NewRatio`.

The `NewSize` and `MaxNewSize` parameters control the new generation's minimum and maximum size. Regulate the new generation size by setting these parameters equal. The bigger the younger generation, the less often minor collections occur. The size of the young generation relative to the old generation is controlled by `NewRatio`. For example, setting `-XX:NewRatio=3` means that the ratio between the old and young generation is 1:3, the combined size of eden and the survivor spaces will be fourth of the heap.

By default, the GlassFish Server is invoked with the Java HotSpot Server JVM. The default `NewRatio` for the Server JVM is 2: the old generation occupies 2/3 of the heap while the new generation occupies 1/3. The larger new generation can accommodate many more short-lived objects, decreasing the need for slow major collections. The old generation is still sufficiently large enough to hold many long-lived objects.

To size the Java heap:

- Decide the total amount of memory you can afford for the JVM. Accordingly, graph your own performance metric against young generation sizes to find the best setting.
- Make plenty of memory available to the young generation. The default is calculated from `NewRatio` and the `-Xmx` setting.

- Larger eden or younger generation spaces increase the spacing between full GCs. But young space collections could take a proportionally longer time. In general, keep the eden size between one fourth and one third the maximum heap size. The old generation must be larger than the new generation.

For up-to-date defaults, see Java HotSpot VM Options (http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html).

Example 4-1 Heap Configuration on Solaris

This is an exmple heap configuration used by GlassFish Server on Solaris for large applications:

```
-Xms3584m
 -Xmx3584m
 -verbose:gc
 -Dsun.rmi.dgc.client.gcInterval=3600000
```

Survivor Ratio Sizing

The `SurvivorRatio` parameter controls the size of the two survivor spaces. For example, `-XX:SurvivorRatio=6` sets the ratio between each survivor space and eden to be 1:6, each survivor space will be one eighth of the young generation. The default for Solaris is 32. If survivor spaces are too small, copying collection overflows directly into the old generation. If survivor spaces are too large, they will be empty. At each GC, the JVM determines the number of times an object can be copied before it is tenured, called the tenure threshold. This threshold is chosen to keep the survivor space half full.

Use the option `-XX:+PrintTenuringDistribution` to show the threshold and ages of the objects in the new generation. It is useful for observing the lifetime distribution of an application.

Rebasing DLLs on Windows

When the JVM initializes, it tries to allocate its heap using the `-Xms` setting. The base addresses of GlassFish Server DLLs can restrict the amount of contiguous address space available, causing JVM initialization to fail. The amount of contiguous address space available for Java memory varies depending on the base addresses assigned to the DLLs. You can increase the amount of contiguous address space available by rebasing the GlassFish Server DLLs.

To prevent load address collisions, set preferred base addresses with the rebase utilty that comes with Visual Studio and the Platform SDK. Use the rebase utility to reassign the base addresses of the GlassFish Server DLLs to prevent relocations at load time and increase the available process memory

for the Java heap.

There are a few GlassFish Server DLLs that have non-default base addresses that can cause collisions. For example:

- The `nspr` libraries have a preferred address of 0x30000000.
- The `icu` libraries have the address of 0x4A?00000.

Move these libraries near the system DLLs (`msvcrt.dll` is at `0x78000000`) to increase the available maximum contiguous address space substantially. Since rebasing can be done on any DLL, rebase to the DLLs after installing the GlassFish Server.

To rebase the GlassFish Server's DLLs

Before You Begin

To perform rebasing, you need:

- Windows 2000
- Visual Studio and the Microsoft Framework SDK rebase utility

1. Make as-install\ `bin` the default directory.

```
cd as-install\bin
```

1. Enter this command:

```
rebase -b 0x6000000 *.dll
```

1. Use the `dependencywalker` utility to make sure the DLLs were rebased correctly.
   For more information, see the Dependency Walker website (`http://www.dependencywalker.com`).
2. Increase the size for the Java heap, and set the JVM Option accordingly on the JVM Settings page in the Admin Console.
3. Restart the GlassFish Server.

Example 4-2 Heap Configuration on Windows

This is an example heap configuration used by Oracle GlassFish Server for heavy server-centric applications, on Windows, as set in the `domain.xml` file.

```
<jvm-options> -Xms1400m </jvm-options>
<jvm-options> -Xmx1400m </jvm-options>
```

See Also

For more information on rebasing, see MSDN documentation for rebase utility (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/rebase.asp).

Further Information

For more information on tuning the JVM, see:

- Java HotSpot VM Options (http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html)
- Frequently Asked Questions About the Java HotSpot Virtual Machine (http://www.oracle.com/technetwork/java/hotspotfaq-138619.html)
- Performance Documentation for the Java HotSpot VM (http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html)
- Java performance web page (http://java.sun.com/javase/technologies/performance.jsp)
- Monitoring and Managing Java SE 6 Platform Applications (http://java.sun.com/developer/technicalArticles/J2SE/monitoring/)
- The jvmstat monitoring utility (http://java.sun.com/performance/jvmstat/)

5 Tuning the Operating System and Platform

This chapter discusses tuning the operating system (OS) for optimum performance. It discusses the following topics:

Server Scaling

This section provides recommendations for optimal performance scaling server for the following server subsystems:

Processors

The GlassFish Server automatically takes advantage of multiple CPUs. In general, the effectiveness of multiple CPUs varies with the operating system and the workload, but more processors will generally improve dynamic content performance.

Static content involves mostly input/output (I/O) rather than CPU activity. If the server is tuned properly, increasing primary memory will increase its content caching and thus increase the relative amount of time it spends in I/O versus CPU activity. Studies have shown that doubling the number of CPUs increases servlet performance by 50 to 80 percent.

Memory

See the section Hardware and Software Requirements in the GlassFish Server Release Notes for specific memory recommendations for each supported operating system.

Disk Space

It is best to have enough disk space for the OS, document tree, and log files. In most cases 2GB total is sufficient.

Put the OS, swap/paging file, GlassFish Server logs, and document tree each on separate hard drives. This way, if the log files fill up the log drive, the OS does not suffer. Also, its easy to tell if the OS paging file is causing drive activity, for example.

OS vendors generally provide specific recommendations for how much swap or paging space to allocate. Based on Oracle testing, GlassFish Server performs best with swap space equal to RAM, plus enough to map the document tree.

Networking

To determine the bandwidth the application needs, determine the following values:

- The number of peak concurrent users (N_{peak}) the server needs to handle.
- The average request size on your site, r. The average request can include multiple documents. When in doubt, use the home page and all its associated files and graphics.
- Decide how long, t, the average user will be willing to wait for a document at peak utilization.

Then, the bandwidth required is:

$N_{peak}r / t$

For example, to support a peak of 50 users with an average document size of 24 Kbytes, and transferring each document in an average of 5 seconds, requires 240 Kbytes (1920 Kbit/s). So the site needs two T1 lines (each 1544 Kbit/s). This bandwidth also allows some overhead for growth.

The server's network interface card must support more than the WAN to which it is connected. For example, if you have up to three T1 lines, you can get by with a 10BaseT interface. Up to a T3 line (45 Mbit/s), you can use 100BaseT. But if you have more than 50 Mbit/s of WAN bandwidth, consider configuring multiple 100BaseT interfaces, or look at Gigabit Ethernet technology.

UDP Buffer Sizes

GlassFish Server uses User Datagram Protocol (UDP) for the transmission of multicast messages to GlassFish Server instances in a cluster. For peak performance from a GlassFish Server cluster that uses UDP multicast, limit the need to retransmit UDP messages. To limit the need to retransmit UDP messages, set the size of the UDP buffer to avoid excessive UDP datagram loss.

To Determine an Optimal UDP Buffer Size

The size of UDP buffer that is required to prevent excessive UDP datagram loss depends on many factors, such as:

- The number of instances in the cluster
- The number of instances on each host
- The number of processors
- The amount of memory
- The speed of the hard disk for virtual memory

If only one instance is running on each host in your cluster, the default UDP buffer size should suffice. If several instances are running on each host, determine whether the UDP buffer is large enough by testing for the loss of UDP packets.

> Note:
>
> On Linux systems, the default UDP buffer size might be insufficient even if only one instance is running on each host. In this situation, set the UDP buffer size as explained in To Set the UDP Buffer Size on Linux Systems.

1. Ensure that no GlassFish Server clusters are running.
   If necessary, stop any running clusters as explained in "To Stop a Cluster" in GlassFish Server Open Source Edition High Availability Administration Guide.
2. Determine the absolute number of lost UDP packets when no clusters are running.
   How you determine the number of lost packets depends on the operating system. For example:
   - On Linux systems, use the `netstat -su` command and look for the `packet receive errors` count in the `Udp` section.
   - On AIX systems, use the `netstat -s` command and look for the `fragments dropped (dup or out of space)` count in the `ip` section.
3. Start all the clusters that are configured for your installation of GlassFish Server.
   Start each cluster as explained in "To Start a Cluster" in GlassFish Server Open Source Edition High Availability Administration Guide.

4. Determine the absolute number of lost UDP packets after the clusters are started.
5. If the difference in the number of lost packets is significant, increase the size of the UDP buffer.

To Set the UDP Buffer Size on Linux Systems

On Linux systems, a default UDP buffer size is set for the client, but not for the server. Therefore, on Linux systems, the UDP buffer size might have to be increased. Setting the UDP buffer size involves setting the following kernel parameters:

- `net.core.rmem_max`
- `net.core.wmem_max`
- `net.core.rmem_default`
- `net.core.wmem_default`

Set the kernel parameters in the `/etc/sysctl.conf` file or at runtime.

If you set the parameters in the `/etc/sysctl.conf` file, the settings are preserved when the system is rebooted. If you set the parameters at runtime, the settings are not preserved when the system is rebooted.

- To set the parameters in the `/etc/sysctl.conf` file, add or edit the following lines in the file:

```
net.core.rmem_max=rmem-max
net.core.wmem_max=wmem-max
net.core.rmem_default=rmem-default
net.core.wmem_default=wmem-default
```

- To set the parameters at runtime, use the sysctl command.

```
$ /sbin/sysctl -w net.core.rmem_max=rmem-max \
net.core.wmem_max=wmem-max \
net.core.rmem_default=rmem-default \
net.core.wmem_default=wmem-default
```

Example 5-1 Setting the UDP Buffer Size in the `/etc/sysctl.conf` File

This example shows the lines in the `/etc/sysctl.conf` file for setting the kernel parameters for controlling the UDP buffer size to 524288.

```
net.core.rmem_max=524288
net.core.wmem_max=524288
net.core.rmem_default=524288
net.core.wmem_default=524288
```

Example 5-2 Setting the UDP Buffer Size at Runtime

This example sets the kernel parameters for controlling the UDP buffer size to 524288 at runtime.

```
$ /sbin/sysctl -w net.core.rmem_max=524288 \
net.core.wmem_max=52428 \
net.core.rmem_default=52428 \
net.core.wmem_default=524288
net.core.rmem_max = 524288
net.core.wmem_max = 52428
net.core.rmem_default = 52428
net.core.wmem_default = 524288
```

Solaris 10 Platform-Specific Tuning Information

Solaris Dynamic Tracing (DTrace) is a comprehensive dynamic tracing framework for the Solaris Operating System (OS). You can use the DTrace Toolkit to monitor the system. The DTrace Toolkit is available through the OpenSolaris project from the DTraceToolkit page (http://hub.opensolaris.org/bin/view/Community+Group+dtrace/dtracetoolkit).

Tuning for the Solaris OS

- Tuning Parameters
- File Descriptor Setting

Tuning Parameters

Tuning Solaris TCP/IP settings benefits programs that open and close many sockets. Since the GlassFish Server operates with a small fixed set of connections, the performance gain might not be significant.

The following table shows Solaris tuning parameters that affect performance and scalability benchmarking. These values are examples of how to tune your system for best performance.

Table 5-1 Tuning Parameters for Solaris

Parameter	Scope	Default	Tuned Value	Comments
rlim_fd_max	/etc/system	65536	65536	Limit of process open file descriptors. Set to account for expected load (for associated sockets, files, and pipes if any).
rlim_fd_cur	/etc/system	1024	8192	+
sq_max_size	/etc/system	2	0	Controls streams driver queue size; setting to 0 makes it infinite so the performance runs won't be hit by lack of buffer space. Set on clients too. Note that setting sq_max_size to 0 might not be optimal for production systems with high network traffic.
tcp_close_wait_interval	ndd /dev/tcp	240000	60000	Set on clients too.
tcp_time_wait_interval	ndd /dev/tcp	240000	60000	Set on clients too.
tcp_conn_req_max_q	ndd /dev/tcp	128	1024	+

Parameter	Scope	Default	Tuned Value	Comments
tcp_conn_req_max_q0	ndd /dev /tcp	1024	4096	+
tcp_ip_abort_interval	ndd /dev /tcp	480000	60000	+
tcp_keepalive_interval	ndd /dev /tcp	7200000	900000	For high traffic web sites, lower this value.
tcp_rexmit_interval_initial	ndd /dev /tcp	3000	3000	If retransmission is greater than 30-40%, you should increase this value.
tcp_rexmit_interval_max	ndd /dev /tcp	240000	10000	+
tcp_rexmit_interval_min	ndd /dev /tcp	200	3000	+
tcp_smallest_anon_port	ndd /dev /tcp	32768	1024	Set on clients too.
tcp_slow_start_initial	ndd /dev /tcp	1	2	Slightly faster transmission of small amounts of data.
tcp_xmit_hiwat	ndd /dev /tcp	8129	32768	Size of transmit buffer.
tcp_recv_hiwat	ndd /dev /tcp	8129	32768	Size of receive buffer.
tcp_conn_hash_size	ndd /dev /tcp	512	8192	Size of connection hash table. See Sizing the Connection Hash Table.

Sizing the Connection Hash Table

The connection hash table keeps all the information for active TCP connections. Use the following command to get the size of the connection hash table:

```
ndd -get /dev/tcp tcp_conn_hash
```

This value does not limit the number of connections, but it can cause connection hashing to take longer. The default size is 512.

To make lookups more efficient, set the value to half of the number of concurrent TCP connections that are expected on the server. You can set this value only in `/etc/system`, and it becomes effective at boot time.

Use the following command to get the current number of TCP connections.

```
netstat -nP tcp|wc -l
```

File Descriptor Setting

On the Solaris OS, setting the maximum number of open files property using `ulimit` has the biggest impact on efforts to support the maximum number of RMI/IIOP clients.

To increase the hard limit, add the following command to `/etc/system` and reboot it once:

```
set rlim_fd_max = 8192
```

Verify this hard limit by using the following command:

```
ulimit -a -H
```

Once the above hard limit is set, increase the value of this property explicitly (up to this limit) using the following command:

```
ulimit -n 8192
```

Verify this limit by using the following command:

```
ulimit -a
```

For example, with the default `ulimit` of 64, a simple test driver can support only 25 concurrent clients, but with `ulimit` set to 8192, the same test driver can support 120 concurrent clients. The test driver spawned multiple threads, each of which performed a JNDI lookup and repeatedly called the same

business method with a think (delay) time of 500 ms between business method calls, exchanging data of about 100 KB. These settings apply to RMI/IIOP clients on the Solaris OS.

Tuning for Solaris on x86

The following are some options to consider when tuning Solaris on x86 for GlassFish Server:

Some of the values depend on the system resources available. After making any changes to `/etc/system`, reboot the machines.

File Descriptors

Add (or edit) the following lines in the `/etc/system` file:

```
set rlim_fd_max=65536
set rlim_fd_cur=65536
set sq_max_size=0
set tcp:tcp_conn_hash_size=8192
set autoup=60
set pcisch:pci_stream_buf_enable=0
```

These settings affect the file descriptors.

IP Stack Settings

Add (or edit) the following lines in the `/etc/system` file:

```
set ip:tcp_squeue_wput=1
set ip:tcp_squeue_close=1
set ip:ip_squeue_bind=1
set ip:ip_squeue_worker_wait=10
set ip:ip_squeue_profile=0
```

These settings tune the IP stack.

To preserve the changes to the file between system reboots, place the following changes to the default TCP variables in a startup script that gets executed when the system reboots:

```
ndd -set /dev/tcp tcp_time_wait_interval 60000
ndd -set /dev/tcp tcp_conn_req_max_q 16384
ndd -set /dev/tcp tcp_conn_req_max_q0 16384
ndd -set /dev/tcp tcp_ip_abort_interval 60000
ndd -set /dev/tcp tcp_keepalive_interval 7200000
ndd -set /dev/tcp tcp_rexmit_interval_initial 4000
ndd -set /dev/tcp tcp_rexmit_interval_min 3000
ndd -set /dev/tcp tcp_rexmit_interval_max 10000
ndd -set /dev/tcp tcp_smallest_anon_port 32768
ndd -set /dev/tcp tcp_slow_start_initial 2
ndd -set /dev/tcp tcp_xmit_hiwat 32768
ndd -set /dev/tcp tcp_recv_hiwat 32768
```

Tuning for Linux platforms

To tune for maximum performance on Linux, you need to make adjustments to the following:

Startup Files

The following parameters must be added to the `/etc/rc.d/rc.local` file that gets executed during system startup.

```
<-- begin
#max file count updated ~256 descriptors per 4Mb.
Specify number of file descriptors based on the amount of system RAM.
echo "6553"> /proc/sys/fs/file-max
#inode-max 3-4 times the file-max
#file not present!!!!!
#echo"262144"> /proc/sys/fs/inode-max
#make more local ports available
echo 1024 25000> /proc/sys/net/ipv4/ip_local_port_range
#increase the memory available with socket buffers
echo 2621143> /proc/sys/net/core/rmem_max
echo 262143> /proc/sys/net/core/rmem_default
#above configuration for 2.4.X kernels
echo 4096 131072 262143> /proc/sys/net/ipv4/tcp_rmem
echo 4096 13107262143> /proc/sys/net/ipv4/tcp_wmem
#disable "RFC2018 TCP Selective Acknowledgements," and
"RFC1323 TCP timestamps" echo 0> /proc/sys/net/ipv4/tcp_sack
echo 0> /proc/sys/net/ipv4/tcp_timestamps
#double maximum amount of memory allocated to shm at runtime
echo "67108864"> /proc/sys/kernel/shmmax
#improve virtual memory VM subsystem of the Linux
echo "100 1200 128 512 15 5000 500 1884 2"> /proc/sys/vm/bdflush
#we also do a sysctl
sysctl -p /etc/sysctl.conf
-- end -->
```

Additionally, create an `/etc/sysctl.conf` file and append it with the following values:

```
<-- begin
 #Disables packet forwarding
net.ipv4.ip_forward = 0
#Enables source route verification
net.ipv4.conf.default.rp_filter = 1
#Disables the magic-sysrq key
kernel.sysrq = 0
fs.file-max=65536
vm.bdflush = 100 1200 128 512 15 5000 500 1884 2
net.ipv4.ip_local_port_range = 1024 65000
net.core.rmem_max= 262143
net.core.rmem_default = 262143
net.ipv4.tcp_rmem = 4096 131072 262143
net.ipv4.tcp_wmem = 4096 131072 262143
net.ipv4.tcp_sack = 0
net.ipv4.tcp_timestamps = 0
kernel.shmmax = 67108864
```

File Descriptors

You may need to increase the number of file descriptors from the default. Having a higher number of file descriptors ensures that the server can open sockets under high load and not abort requests coming in from clients.

Start by checking system limits for file descriptors with this command:

```
cat /proc/sys/fs/file-max
8192
```

The current limit shown is 8192. To increase it to 65535, use the following command (as root):

```
echo "65535"> /proc/sys/fs/file-max
```

To make this value to survive a system reboot, add it to `/etc/sysctl.conf` and specify the maximum number of open files permitted:

```
fs.file-max = 65535
```

Note that the parameter is not `proc.sys.fs.file-max`, as one might expect.

To list the available parameters that can be modified using `sysctl`:

```
sysctl -a
```

To load new values from the `sysctl.conf` file:

```
sysctl -p /etc/sysctl.conf
```

To check and modify limits per shell, use the following command:

```
limit
```

The output will look something like this:

```
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    0 kbytes
memoryuse       unlimited
descriptors     1024
memorylocked    unlimited
maxproc         8146
openfiles       1024
```

The `openfiles` and `descriptors` show a limit of 1024. To increase the limit to 65535 for all users, edit `/etc/security/limits.conf` as root, and modify or add the `nofile` setting (number of file) entries:

```
*         soft    nofile                  65535
*         hard    nofile                  65535
```

The character "*" is a wildcard that identifies all users. You could also specify a user ID instead.

Then edit `/etc/pam.d/login` and add the line:

```
session required /lib/security/pam_limits.so
```

On Red Hat, you also need to edit `/etc/pam.d/sshd` and add the following line:

```
session required /lib/security/pam_limits.so
```

On many systems, this procedure will be sufficient. Log in as a regular user and try it before doing the remaining steps. The remaining steps might not be required, depending on how pluggable authentication modules (PAM) and secure shell (SSH) are configured.

Virtual Memory

To change virtual memory settings, add the following to `/etc/rc.local`:

```
echo 100 1200 128 512 15 5000 500 1884 2> /proc/sys/vm/bdflush
```

For more information, view the man pages for `bdflush`.

Network Interface

To ensure that the network interface is operating in full duplex mode, add the following entry into `/etc/rc.local`:

```
mii-tool -F 100baseTx-FD eth0
```

where eth0 is the name of the network interface card (NIC).

Disk I/O Settings

To tune disk I/O performance for non SCSI disks

1. Test the disk speed.
   Use this command:

```
/sbin/hdparm -t /dev/hdX
```

1. Enable direct memory access (DMA).
   Use this command:

```
/sbin/hdparm -d1 /dev/hdX
```

1. Check the speed again using the `hdparm` command.
   Given that DMA is not enabled by default, the transfer rate might have improved considerably. In order to do this at every reboot, add the `/sbin/hdparm -d1 /dev/hdX` line to `/etc/conf.d/local.start`, `/etc/init.d/rc.local`, or whatever the startup script is called.
   For information on SCSI disks, see: System Tuning for Linux Servers — SCSI (`http://people.redhat.com/alikins/system_tuning.html#scsi`).

TCP/IP Settings

To tune the TCP/IP settings

1. Add the following entry to `/etc/rc.local`

```
echo 30> /proc/sys/net/ipv4/tcp_fin_timeout
echo 60000> /proc/sys/net/ipv4/tcp_keepalive_time
echo 15000> /proc/sys/net/ipv4/tcp_keepalive_intvl
echo 0> /proc/sys/net/ipv4/tcp_window_scaling
```

1. Add the following to `/etc/sysctl.conf`

```
# Disables packet forwarding
net.ipv4.ip_forward = 0
# Enables source route verification
net.ipv4.conf.default.rp_filter = 1
# Disables the magic-sysrq key
kernel.sysrq = 0
net.ipv4.ip_local_port_range = 1204 65000
net.core.rmem_max = 262140
net.core.rmem_default = 262140
net.ipv4.tcp_rmem = 4096 131072 262140
net.ipv4.tcp_wmem = 4096 131072 262140
net.ipv4.tcp_sack = 0
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_window_scaling = 0
net.ipv4.tcp_keepalive_time = 60000
net.ipv4.tcp_keepalive_intvl = 15000
net.ipv4.tcp_fin_timeout = 30
```

1. Add the following as the last entry in `/etc/rc.local`

```
sysctl -p /etc/sysctl.conf
```

1. Reboot the system.
2. Use this command to increase the size of the transmit buffer:

```
tcp_recv_hiwat ndd /dev/tcp 8129 32768
```

Tuning UltraSPARC CMT-Based Systems

Use a combination of tunable parameters and other parameters to tune UltraSPARC CMT-based systems. These values are an example of how you might tune your system to achieve the desired result.

Tuning Operating System and TCP Settings

The following table shows the operating system tuning for Solaris 10 used when benchmarking for performance and scalability on UtraSPARC CMT-based systems (64-bit systems).

Table 5-2 Tuning 64-bit Systems for Performance Benchmarking

Parameter	Scope	Default Value	Tuned Value	Comments
`rlim_fd_max`	`/etc/system`	65536	260000	Process open file descriptors limit; should account for the expected load (for the associated sockets, files, pipes if any).
`hires_tick`	`/etc/system`	+	1	+

Parameter	Scope	Default Value	Tuned Value	Comments
sq_max_size	/etc/system	2	0	Controls streams driver queue size; setting to 0 makes it infinite so the performance runs won't be hit by lack of buffer space. Set on clients too. Note that setting sq_max_size to 0 might not be optimal for production systems with high network traffic.
ip:ip_squeue_bind	+	+	0	+
ip:ip_squeue_fanout	+	+	1	+
ipge:ipge_taskq_disable	/etc/system	+	0	+
ipge:ipge_tx_ring_size	/etc/system	+	2048	+
ipge:ipge_srv_fifo_depth	/etc/system	+	2048	+
ipge:ipge_bcopy_thresh	/etc/system	+	384	+
ipge:ipge_dvma_thresh	/etc/system	+	384	+
ipge:ipge_tx_syncq	/etc/system	+	1	+
tcp_conn_req_max_q	ndd /dev/tcp	128	3000	+
tcp_conn_req_max_q0	ndd /dev/tcp	1024	3000	+

Parameter	Scope	Default Value	Tuned Value	Comments
tcp_max_buf	ndd /dev /tcp	+	4194304	+
tcp_cwnd_max	ndd/ dev/ tcp	+	2097152	+
tcp_xmit_hiwat	ndd /dev /tcp	8129	400000	To increase the transmit buffer.
tcp_recv_hiwat	ndd /dev /tcp	8129	400000	To increase the receive buffer.

Note that the IPGE driver version is 1.25.25.

Disk Configuration

If HTTP access is logged, follow these guidelines for the disk:

- Write access logs on faster disks or attached storage.
- If running multiple instances, move the logs for each instance onto separate disks as much as possible.
- Enable the disk read/write cache. Note that if you enable write cache on the disk, some writes might be lost if the disk fails.
- Consider mounting the disks with the following options, which might yield better disk performance: `nologging`, `directio`, `noatime`.

Network Configuration

If more than one network interface card is used, make sure the network interrupts are not all going to the same core. Run the following script to disable interrupts:

```
allpsr=`/usr/sbin/psrinfo | grep -v off-line | awk '{ print $1 }'`
   set $allpsr
   numpsr=$#
   while [ $numpsr -gt 0 ];
   do
       shift
       numpsr=`expr $numpsr - 1`
       tmp=1
       while [ $tmp -ne 4 ];
       do
           /usr/sbin/psradm -i $1
           shift
           numpsr=`expr $numpsr - 1`
           tmp=`expr $tmp + 1`
       done
   done
```

Put all network interfaces into a single group. For example:

```
$ifconfig ipge0 group webserver
$ifconfig ipge1 group webserver
```