

# Optimize GlassFish Performance in a Production Environment

Performance White Paper  
February 2009

## Abstract

Sun GlassFish Application Server v2 is a high performance application server. This white paper will provide some guidance on how to tune the application server and is intended to be used as an introductory step to achieving better performance for your application. Eleven tuning parameters will be described with some general recommendations. These recommendations can be used to optimize the performance of the application server. These recommendations are followed by data from a benchmarking exercise that demonstrates the effectiveness of the tuning changes.



# Table of Contents

Tip 1: Java Version.....	4
Tip 2: JVM Mode.....	4
Tip 3: Java Heap Size.....	4
Tip 4: Tune Java Garbage Collection.....	5
Tip 5: HTTP acceptor threads.....	5
Tip 6: HTTP request processing threads.....	5
Tip 7: Keep Alive subsystem.....	6
Tip 8: HTTP File Cache.....	6
Tip 9: Default-web.xml.....	7
Tip 10: JDBC Tuning.....	8
Tip 11: Disable Access Logging.....	9
Specific Tuning for CoolThreads™ Technology on Systems with UltraSPARC T1/T2 Processors System Specific Tuning.....	9

References.....	11
-----------------	----

Appendix I.....	13
-----------------	----

Tip 1 - Changing Java version.....	13
Tips 2-4 - Editing/Adding JVM options.....	13
Tip 5 - HTTP Acceptor Threads.....	14
Tip 6 - HTTP Request Processing Threads.....	16
Tip 7 – Keep Alive settings.....	17
Tip 8 – HTTP File Cache.....	18
Tip 9 – Edit the default-web.xml.....	21
Tip 10 – JDBC Tuning.....	21
Tip 11 – Disable Access Logging.....	22
Tip for CoolThreads UltraSPARC T1/T2 Tuning.....	23
Rotating access log.....	23
Use libumem.....	23
Use of FX scheduling class.....	23
Network Configuration.....	23

## Executive Summary

This white paper will describe and provide general guidance for initial tuning values for eleven parameters in order to improve your application performance on the Sun GlassFish™ application server v2. Additional specific tuning advice is also provided for the CoolThreads UltraSPARC T1/T2 platform. While optimal values may be different for each deployment of the GlassFish application server, it is useful to provide these first eleven steps as an initial primer to benchmarking your own application.

## Introduction

Obtaining optimum performance from your application deployed on the GlassFish application server requires a thorough understanding of your objectives. For some, best performance is attained with maximum throughput which is defined as the number of transactions completed per second. For others, best performance may be the maximum number of users supported on a system concurrently within a desired response time. Unfortunately, there are no universal tuning parameters you can set that will reap the best performance. There are many factors to consider such as the type of application, operating system, Java version, system hardware architecture, deployment architecture, number of users, expected response time, and network bandwidth. While all of these variables may initially appear daunting, the aim of this whitepaper is to reduce the complexity and provide some initial steps to consider to achieve your performance goals on the GlassFish application server.

## Methodology

Before embarking on a tuning exercise, one of the most important steps is to choose your application and testing environment appropriately. Too often, one rushes to apply tunings to improve performance without spending enough time to analyze the problem, the desired metrics and a thorough scrutiny of the collected data. This methodology includes:

- choosing the appropriate application: In general, the actual application used in production is a good place to start. If this application is too complex, then it may be necessary to write a microbenchmark that is representative of the most common scenario. It is worthwhile to spend some time analyzing what the most common usage patterns are in your application. This seems obvious but

choosing the incorrect benchmark can lead to misleading results about software and hardware requirements and ultimately a waste of time and money. Continuous evaluation of what you are testing versus what you have in production will avoid the red herrings. Selecting the incorrect application to benchmark is akin to choosing a fork to drink soup.

- planning your test environment: Ideally, it is best to have separate system for driving the load and another for your 'system under test', or SUT on a dedicated separate network so that the network bandwidth and traffic are not limiting factors. If this is not possible, you can create separate virtual containers (eg. Solaris Zones) or different processor sets. For example, you could have the driver and SUT on the same machine but create two separate processor sets to monitor the usage of the driver and SUT resources separately.



Figure 1. Typical benchmarking setup with driver, SUT and DB on separate machines.

- Understand the metric being measured. For example, solely looking at throughput could be misleading without taking into account the response time. Typically, a threshold value of response time is defined as a passing criterion. If the measured response time exceed this value, the transaction is deemed as a failure. A transaction in this context represents a scenario (or many scenarios) that you are testing, such as adding an item to a shopping cart or logging out. Since the average response time of all the transactions can be skewed by a few outlying data points, a typical statistical measure used is the max 90<sup>th</sup> percentile. For example, suppose that the criteria for response time for a particular transaction is 2 seconds. The overall test is considered a success if the response times are 2 seconds or below for 90% of all recorded transactions.

A necessary component is investigating the load testing tool you are using – both the behavior of the load generating client and the statistics being reported. Occasionally, the client can be found to be the bottleneck in benchmarking and consequently cannot place enough load on your system under test. Also, spend some time not only analyzing the measurements, but how the measurements are reported by the tool. There are many good open source and commercial tools available; one worthy of mention is [Faban](#), an

open-source driver and harness framework.

- Plan how the test is carried out: Typically, benchmarking can be divided into three phases. The first, the warm up period, is used to increase the client load by adding more concurrent threads. This warm up period should be long enough for the server JVM™ to warm up and allow it to finish the JIT compiler optimizations. The measurement of the system under test should occur during the second period, the steady state. This should be long enough so that the data collected during this period is representative sample of the test. Each benchmark will have a degree of randomness with respect to the metric reported so the steady state duration needs to be long enough to minimize the randomness. The final phase, the ramp down, allows each client thread to end gracefully. Consideration should also be made of the 'think time' – the period of time between the end of one request and the beginning of the new one.
- Monitor basic statistics during your benchmarking test. Obtaining a number at the end without knowing how your system behaved during the test is not meaningful. Collecting system statistics like CPU utilization, network and disk IO statistics during your test can serve as an excellent source of information.

This introduction of benchmarking methodology is by no means complete, but hopefully it can serve as a quick primer on how to get started.

## Top 11 Tuning Parameters for GlassFish

This section's title can lead to incorrect assumptions so it is necessary to add a *caveat emptor*.

*“The best tuning tip is the one that improves YOUR application's performance. Your mileage may vary.”*

It is impossible to determine a set of tunings that will improve every application deployed on the GlassFish application server and predict how much your throughput will increase. Benchmarking consists of a repetitive cycle of testing, apply tuning, re-testing so you will need to carry out your experiments. However, faced with all of the knobs to turn, here is a list of the eleven you can start with, in no particular order.

Since Java™ Platform, Standard Edition 5 (Java SE Platform 5) , there is an ergonomics feature where the Java platform will match the choice of garbage collector, heap size and runtime compiler based on the class of the machine. This minimizes the amount of tuning if there are no explicitly set server JVM options. However, since it is impossible to optimally tune for every application with ergonomics, there may be certain parameters than can be tweaked above

and beyond the assumptions made by this feature. Please refer to <http://java.sun.com/docs/hotspot/gc5.0/ergo5.html> for more details.

---

Note: The exact steps of configuring each parameter will be covered in Appendix I.

---

#### Tip 1: Java Version

Typically, many performance optimizations are always being incorporated in newer releases of Java SE. Whenever possible, it is a good idea to upgrade to the latest version of Java SE Platform-to take advantage of these optimizations.

#### Tip 2: JVM Mode

By default, the GlassFish application server is configured to use the client VM. For performance testing, it is recommended to change this to “-server”. So why is the app server set to “-client” by default? Typically, the GlassFish application server is installed in developer profile so by using the -client setting allows for ease of development like faster startup and deployment times. Note that if you are using the GlassFish application server in cluster or enterprise profile, the -server should be the default option.

#### Tip 3: Java Heap Size

The size of the heap is determined by the Java options -Xmx (maximum) and -Xms (minimum). While a larger heap can contain more objects and reduce the frequency of garbage collection, it may result in longer garbage collection times especially for a full GC cycle. The recommendation is to tune the heap based on the size of total available memory in your system, process data model (32-bit or 64-bit) and operating system. The heap size should not be bigger than the physical memory. By default, the GlassFish application server is set to 512 MB and the limit for a 32 bit JVM version- is approximately 3.5GB on Solaris, and 1.5 GB on Windows without any operating system tweaks. For Linux, it depends on the version you are using varying from 1.4 GB to a 3.5 GB heap. For 64 bit process models, the maximum is theoretically unlimited but your performance may degrade, depending on your application, approximately 5-20% in a 64 bit mode.

Another suggestion is to set Xmx and Xms to the same value to avoid unnecessary resizing of the heap. For more information on JVM heap sizes, please refer to <http://java.sun.com/performance/reference/whitepapers/tuning.html#section4.1.2>

#### Tip 4: Tune Java Garbage Collection

This option should be explored if you do some initial GC analysis (eg.

-verbose:gc, -XX:+PrintGCTimeStamps, -XX:+PrintGCDetails) and the JVM machine is spending a lot of time doing garbage collection. There are also visual tools available like jvmstat to connect to a Java 6 process. By virtue of ergonomics, GC tuning is typically not required unless there is an overt GC problem. By default, the serial collector is the default garbage collector and is typically used for single processor machines and a small heap. However, on server-class machines with more than one processor, parallel GC is the default. Ensure that parallel GC is being used (-XX:+UseParallelGC) for multithreaded machines which uses multiple threads for minor collections. Major collections are the same as serial collector. Another JVM tuning does exist for parallel old generation collector (-XX:+UseParallelOldGC) however this is only applicable in only certain phases of an old generation collection. If short response time is more critical to your application, the CMS collector is better suited for short GC pauses at the expense of throughput. (-XX:+UseConcMarkSweepGC). By trading processor resources which would otherwise be available to the application for shorter major collection pause times the concurrent part of the collection is done by a single garbage collection thread. A more thorough understand is necessary when tinkering with GC tuning options and it is strongly recommended to read "[Tuning Garbage Collection with the Java Virtual Machine](#)" for more details and GC strategies.

#### Tip 5: HTTP acceptor threads

HTTP acceptor threads accept new incoming connections and schedule new requests for the existing connections. The default number of acceptor threads is one. It is recommended to have 1 thread per 1-4 core, although experimentation may be necessary to find the optimal number.

#### Tip 6: HTTP request processing threads

This pool of threads retrieve and process incoming HTTP requests. The default number of request processing threads is 5 but a starting rule of thumb is to tune the number of HTTP request processing threads to the number of CPUs on the SUT. If you application is I/O bound, you can start with double the number of CPUs. Increase this number of threads until your throughput starts to decline. At the point when your throughput starts to suffer, the request processing threads are starting to contend for CPU resources so some experimentation will be necessary to find the sweet spot.

#### Tip 7: Keep Alive subsystem

This subsystem prevents the server from being overloaded with connections. A waiting keep alive connection has completed processing the previous request, and is waiting for a new request to arrive on the same connection. Three factors describe how the Keep-Alive behavior. The first factor, the **Keep-Alive timeout**,

defines how long a connection should be maintained since its last activity. The default value is 30 seconds and it may be increased if the typical time between client interactions is greater than 30 seconds to avoid the expense of creating a new connection for a new request. However, you do not want to set a very high value since the cost to the server is maintaining too many open connections. The second factor, the **Keep-Alive max connections**, define the maximum number of requests that would be handled on a connection before the connection is closed by the server. Setting this value to -1 will disable this feature. Again, some experimentation may be necessary.

The third parameter is the **thread-count**, which acts much like the HTTP acceptor thread. A first rule of thumb is to keep one for every 8 cores in your system.

#### Tip 8: HTTP File Cache

If your application contains static files, it is recommended to enable HTTP file cache to optimize performance. The HTTP server cache will put frequently used static files in memory to avoid reading the resource from the file system for every request. Files are categorized based on user-defined size limits into small, medium and large groups. The contents of small files are read into the JVM heap while medium files are memory mapped by GlassFish. Large files (i.e. any file larger than the medium-file-size-limit) are not cached. Additionally, the request has to be for a static resource that is serviced by the Default Servlet. Files with URI mappings to custom servlets will not be cached.

The cost of using file caching is the increased memory footprint of the server instance. Specifically, small file caching will increase garbage collection since they are cached in the JVM heap while memory-mapped medium sized files will increase the resident memory of the process.

The following factors should be taken into consideration while tuning the file cache: the number and size of the various files, the frequency of the file request, the heap space configured for the instance, and the amount of available memory.

- Set the number of files to be cached based on the number of files that are commonly accessed. For most cases, the default value of 1024 will suffice.
- Set the maximum age of a cache entry based on how often clients access a given resource. The `max-age-in-seconds` parameter determines the amount of time a file is retained in the cache. It is recommended that this value be set large enough so as to incur several hits before the file is removed from the cache. Conversely, the value should not be so high to avoid caching of infrequently used files

which can impact performance negatively or for files that change frequently which may show incorrect behavior.

- Set the space available for the small files based on the amount of heap available. It is recommended that this parameter be set to a value large enough to accommodate the commonly accessed files. Setting this value to be a large proportion of the heap can cause frequent garbage collection.
- Set the space available for medium files based on the overall available memory. Since medium files are memory mapped, the memory associated with this cache is external to the JVM heap. Caching these files increase the process memory so it is important to ensure that the overall memory does not go beyond the available process memory (4GB for 32 bit JVM). If the amount of physical memory in the system is limited, it is recommended that the space allocated for the medium sized files be reduced so that the overall process memory is lower than the available physical memory.

#### Tip 9: Default-web.xml

The default-web.xml file defines features such as filters and security constraints that apply to all web applications. The parameter, **development=true**, (the default value for developer profile) enables changes made to JSP™ - code to be instantly visible to the clients. However, there is a cost associated with this. To avoid the cost of checking whether the JSP code has been modified and hence its recompilation, the first parameter, **development=false**, can be used to set development to false since this scenario is unlikely in a production system. This check affects application scalability when multiple users request the same JSP class.

The second parameter, **genStrAsCharArray=true**, changes the way the JSPs are generated by generating char arrays for every static strings in the JSP class like for example, the HTML tags. By default, the JSPcode writer must call the toCharArray() on every String on every invocation of the JSPclass.

Table 1. Performance optimization in default-web.xml.

```
<init-param>
  <param-name>development</param-name>
  <param-value>>false</param-value>
</init-param>
<init-param>
  <param-name>genStrAsCharArray</param-name>
  <param-value>>true</param-value>
</init-param>
```

---

### Tip 10: JDBC Tuning

If your application uses Java DataBase Connectivity ("JDBC™") software for database access, it may be beneficial to tune your database connection pool. Tune the steady-pool-size and the max-pool-size, and set them to the same value. This will avoid unnecessary resizing of the pool during the test. A general rule of thumb is to tune the value for max-pool-size and steady-pool-size to the same number of HTTP request processing threads.

If your JDBC driver supports this feature, it is advisable to use JDBC drivers that use statement caching to re-use prepared statements. Check with your database vendor on how to do so.

For Oracle, you can add the following properties to your JDBC connection pool:

Table 2. Oracle database properties for statement caching

```
<property name="ImplicitCachingEnabled" value="true"/>
<property name="MaxStatements" value="200"/>
```

---

#### For MySQL databases:

Table 3. MySQL database properties for statement caching

```
<property name="cachePrepStmts" value="true"/>
<property name="prepStmtCacheSize" value="512"/>
<property name="useServerPreparedStmts" value="false" />
```

---

#### For DB2 databases:

Table 4. DB2 database properties for statement caching

```
<property name="MaxPooledStatements" value="200"/>
```

---

### Tip 11: Disable Access Logging

**Disable access logging:** To avoid unnecessary I/O activity, disable the access logging.

#### Specific Tuning for CoolThreads™ Technology on Systems with UltraSPARC<sup>R</sup> T1/T2 Processors

For CoolThreads technology on systems with UltraSPARC T1/T2 processors, there are specific tuning parameters that can be applied to take advantage of performance.

- Ensure that the JDK™ version is at least Java SE platform 1.5.0\_06.
- -XX:LargePageSizeInBytes=256m (requires Java SE platform 1.5.0\_06 and Solaris 10 HW2)
- -Xmx2560m -Xms2560m (in general, it is optimal to give the application server as much memory as possible within a 32-bit address space. When using the LargePageSizeInBytes flag, the heap size must be a multiple of that page size. If you must use less memory, ensure that it is a multiple of 256m).
- XX:ParallelGCThreads=4. This is a good starting point but may be needed to be tuned according to GC activity in your application and the number of threads available.
- Separate the access logs, transaction logs and imq files onto as many disks as possible
- Use libumem
- Use FX Scheduling class
- If you are using more than one network interface, then make sure that all the network interrupts are not going to the same core.

Consult the Appendix I for more details.

## Case Study

To illustrate the effect of tuning the GlassFish application server effectively, an e-commerce type web application was deployed on the app server and subjected to a load with varying tunings applied. However, before presenting the data, it is a good opportunity to restate the *caveat emptor*:

*“The best tuning tip is the one that improves YOUR application's performance. Your mileage may vary”.*

You may not observe the same improvement in performance due to the hardware, OS, JDK version and of course, the benchmark application.

Table I summarizes the tuning applied to the web application. Configuration A is the default tuning applied to the GlassFish application server with the JVM option -client and -Xmx 512m. Configuration B keep the same heap size with only the JVM mode set to -server instead of -client. Configuration C increases the heap size to 1.5 GB and Configuration D applies HTTP request processor, HTTP Keep Alive tuning, HTTP Acceptor Thread, JDBC caching and default-web.xml tunings.

Table 5. Summary of Tunings Applied for Configuration A, B, C and D.

Configuration	Description
A	Default GlassFish tuning. JVM options: -client, Xmx 512m
B	JVM options: -server, Xmx512m
C	JVM options: -server, Xmx1500m, 1500m
D	JVM options: -server, Xmx1500m, 1500m HTTP Acceptor Threads, HTTP Request Processing, HTTP KeepAlive, JDBC caching, default-web.xml
E	JVM options: -server, Xmx1500m, Xms1500m HTTP RequestProcessing, HTTP KeepAlive tuning, JDBC tuning

The results obtained were from a single instance of the GlassFish application server demonstrate a marked improvement from tuning many of the parameters outlined in the list described above. Please exercise caution in interpreting these results; the improvement shown is case specific so your results may not follow a similar trend for each tuning applied. In some cases, no improvement or a degradation may even result.

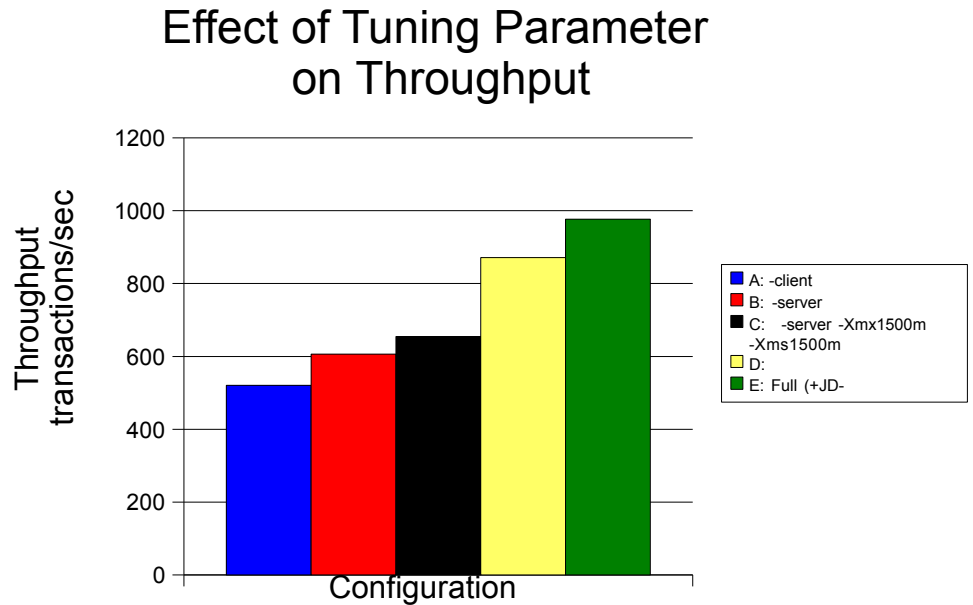


Figure 2: Summary of Effect of Tuning Configuration on Throughput.

## Conclusion

This white paper does not attempt to cover all of the possible tuning parameters in the GlassFish application server; instead it provides an initial guidance on certain parameter for experimenting during benchmarking. Firstly, it is imperative to use the correct application for benchmarking. The experiments should be performed in a controlled environment with a clear understanding of what the key metrics are crucial for your business scenario(s). Each tuning parameter will need to be tested with trial and error in an iterative cycle with system data collected for each experiment. Following good methodology, these ten tips represent your first ten steps towards obtaining the best performance on the GlassFish application server.

## References

1. [GlassFish Performance Tuning Guide](#)
2. [GlassFish Application Deployment Guide](#)
3. [Developing and Tuning Applications on UltraSPARC T1 Chip Multithreading Systems](#)
4. <http://java.sun.com/performance/reference/whitepapers/tuning.htm>

5. [http://java.sun.com/performance/reference/whitepapers/5.0\\_performance.html](http://java.sun.com/performance/reference/whitepapers/5.0_performance.html)
6. [GlassFish Administration Guide](#)
7. <http://faban.sunsource.net/>
8. [Tuning Garbage Collection with the 5.0 Java\[tm\] Virtual Machine](#)
9. [Ergonomics in the 5.0 Java\[tm\] Virtual Machine](#)

## Acknowledgments

Madhu Konda, Scott Oaks, Binu John, Suveen Nadipalli and Deep Singh for reviewing this white paper and the Java Performance Team for the contribution to the content.

## Appendix I

---

Note – A third option is available for most of these tips although it is not a recommended. You can edit the domain.xml directly via your favorite text editor found in \$GLASSFISH\_HOME/domains/domain1/config where \$GLASSFISH\_HOME is the directory where you installed GlassFish. Proceed with caution as GlassFish may not start if your domain.xml is malformed.

---

### Tip 1 - Changing Java version

To change Java version, edit the \$GLASSFISH\_HOME/config/asenv.conf. Change the property called “AS\_JAVA” to point to the desired Java version.

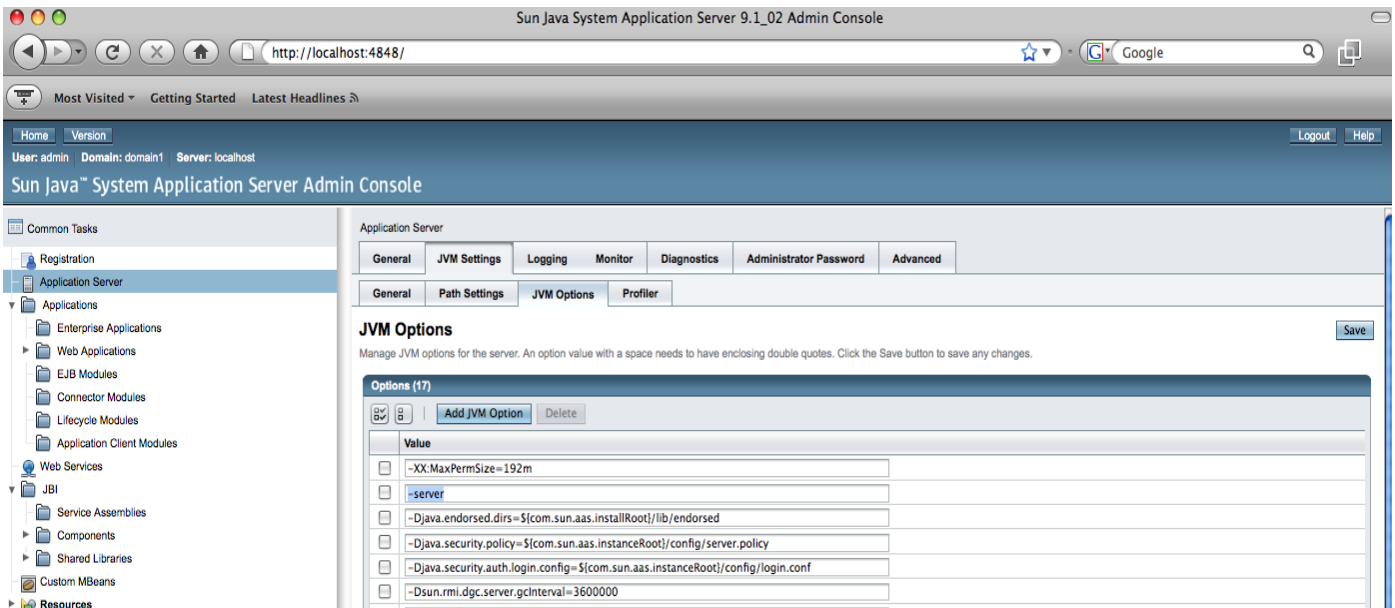
### Tips 2-4 - Editing/Adding JVM options

To apply JVM options, you can use the following options:

Option 1: via Administration console:

1. Use the web browser URL: `http:<yourhostname>:4848` (the default admin port)
2. Login with administrator user and password (default admin/adminadmin)
3. Click on Application Server node on the left hand side, JVM settings tab on the right hand side, then JVM options.
4. Edit the desired JVM option (or add New) in the textbox.
5. Click Save on the right hand side

## 6. Restart GlassFish.



### Option 2: asadmin CLI command

You can also use the CLI `asadmin` command to [list](#), [delete](#) or [create](#) new JVM option:

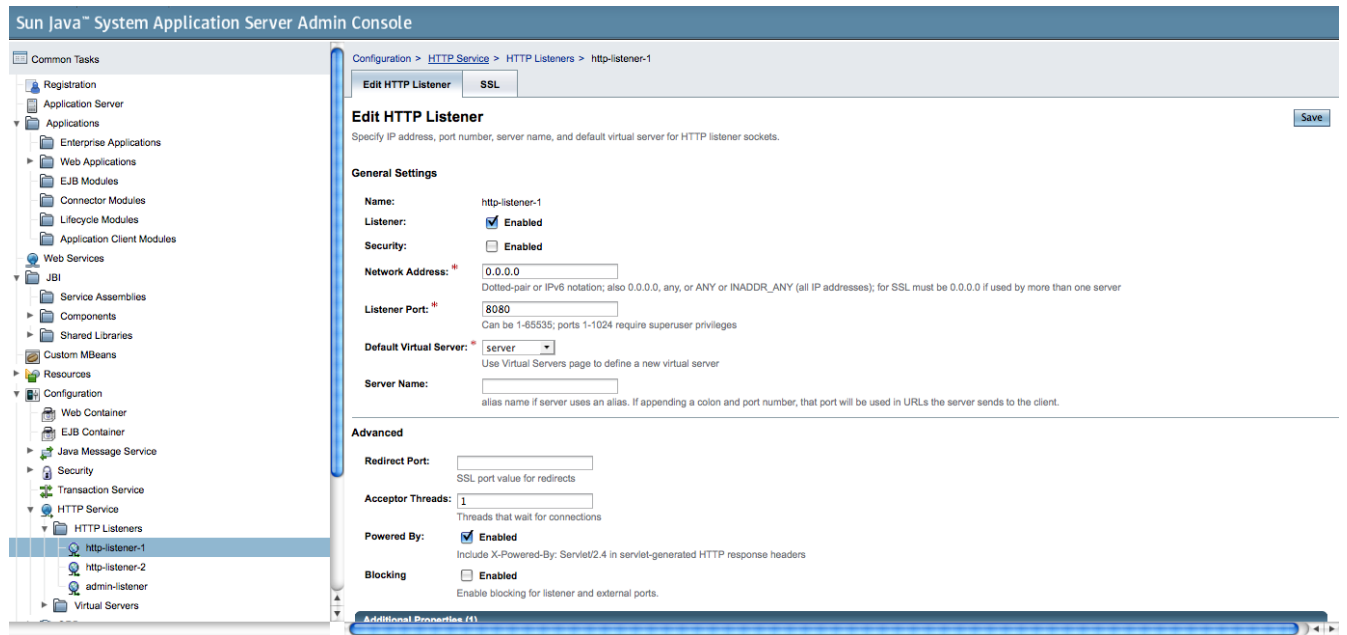
Examples:

```
asadmin create-jvm-options --user admin "-Xmx1024m:"
asadmin delete-jvm-options --user admin "-client:"
asadmin create-jvm-options --user admin "-server:"
```

### Tip 5 - HTTP Acceptor Threads

Option 1 - via Administration console:

1. Login at the administration URL: `http://<yourhostname>:4848`
2. Expand the node on the left hand side by clicking on Configuration --> HTTP Service --> HTTP Listeners
3. Click on `http-listener-1`.
4. Edit the "Acceptor Threads" field under the Advanced setting.



## Option 2: via asadmin CLI

use of the asadmin list/get/set for HTTP properties

Example:

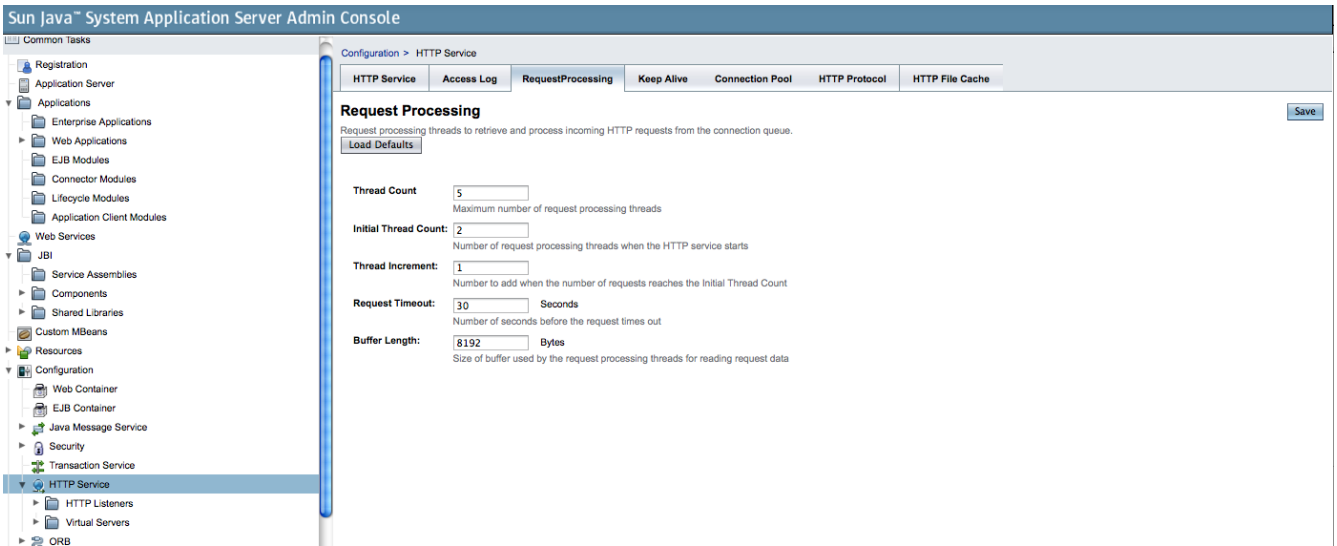
```
asadmin get server.http-service.http-listener.http-listener-1.acceptor-threads
```

```
asadmin set server.http-service.http-listener.http-listener-1.acceptor-threads = <number of threads>
```

## Tip 6 - HTTP Request Processing Threads

Option 1 - via Administration console:

1. Login at the administration URL: `http://<yourhostname>:4848`
2. Expand the node on the left hand side by clicking on Configuration --> HTTP Service
3. Click on RequestProcessing Tab on right hand side.
4. Edit the text box beside Thread Count.
5. Click Save and restart.



### Option 2: via asadmin CLI

use of the asadmin list/get/set for HTTP properties

Example:

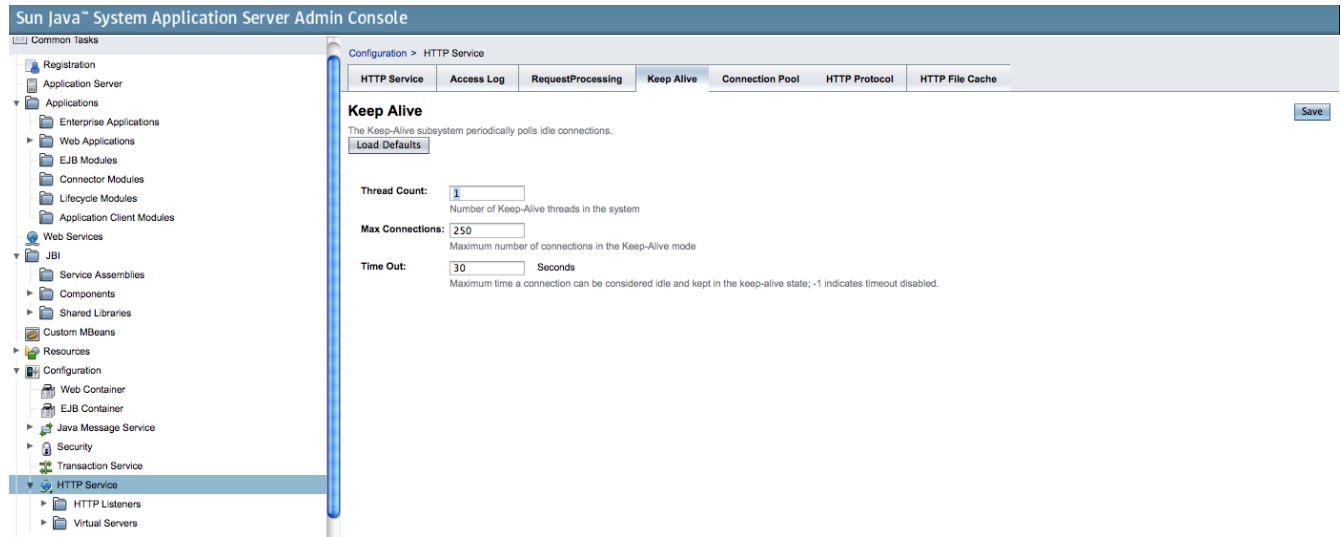
```
asadmin get server.http-service.request-processing.thread-count
```

```
asadmin set server.http-service.request-processing.thread-count = <number of threads>
```

### Tip 7 – Keep Alive settings

Option 1 - via Administration console:

1. Login at the administration URL: `http://<yourhostname>:4848`
2. Expand the node on the left hand side by clicking on Configuration --> HTTP Service
3. Click on KeepAlive Tab on right hand side.
4. Edit Thread Count text box.
5. Click Save and Restart.



### Option 2: via asadmin CLI

use of the [asadmin list/get/set](#) for HTTP properties

Example:

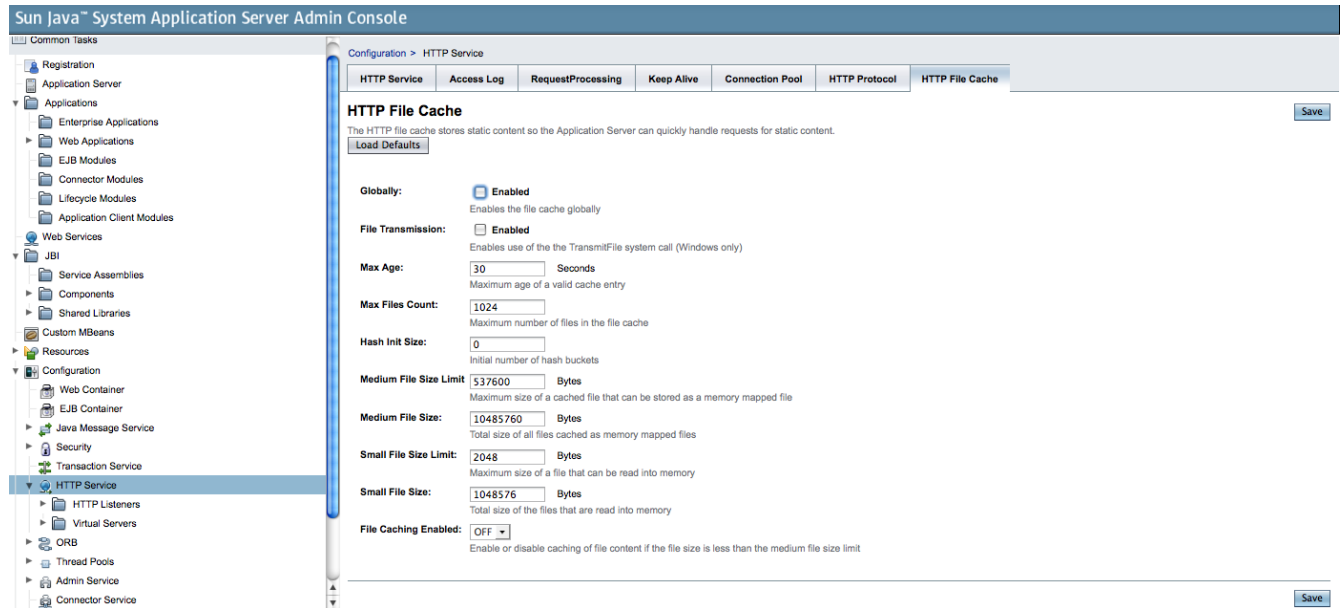
```
asadmin get server.http-service.keep-alive.thread-count
asadmin set server.http-service.keep-alive.thread-count =
<number of threads>
```

### Tip 8 – HTTP File Cache

The first option, enabling file cache globally refers to small file cache (JVM heap) while the last option, “File Cache Enabled” is used to enable medium file cache (m-mapped files).

Option 1 - via Administration console:

1. Login at the administration URL: `http://<yourhostname>:4848`
2. Expand the node on the left hand side by clicking on Configuration --> HTTP Service
3. Click on HTTP File Cache Tab on right hand side.
4. Click on the “enabled” check box.
5. Click Save and restart.



## Option 2: via asadmin CLI

use of the `asadmin list/get/set` for HTTP properties

Example:

Small File caching:

```
asadmin get server.http-service.http-file-cache.file-caching-enabled
```

```
asadmin set server.http-service.http-file-cache.file-caching-enabled = true
```

Medium File caching:

```
asadmin get server.http-service.http-file-cache.globally-enabled
```

```
asadmin set server.http-service.http-file-cache.globally-enabled = true
```

Other properties:

```
server.http-service.http-file-cache.file-transmission-enabled (Windows only)
```

```
server.http-service.http-file-cache.hash-init-size
```

```
server.http-service.http-file-cache.max-age-in-seconds
```

```
server.http-service.http-file-cache.max-files-count
```

```
server.http-service.http-file-cache.medium-file-size-limit-in-bytes
```

```
server.http-service.http-file-cache.medium-file-space-in-bytes
```

```
server.http-service.http-file-cache.small-file-size-limit-in-bytes
```

```
server.http-service.http-file-cache.small-file-space-in-bytes
```

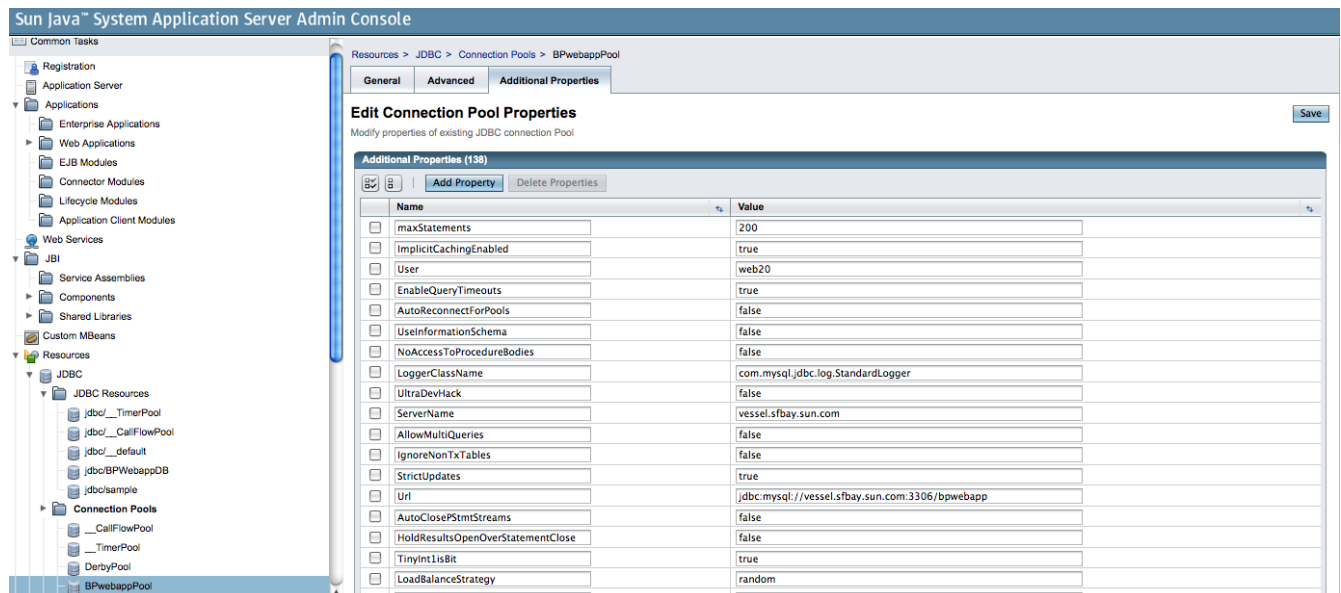
### Tip 9 – Edit the default-web.xml

With your favorite text editor , edit the default-web.xml found in the directory \$GLASSFISH\_HOME/domains/domain1/config.

### Tip 10 – JDBC Tuning

Option 1 - via Administration console:

1. Login at the administration URL: `http://<yourhostname>:4848`
2. Expand the node on the left hand side by clicking on Resource--> JDBC --> Connection Pools
3. Click on <myConnectionPoolName>
4. Click on Additional Properties on right hand side.
5. Click on “Add Properties” button
6. Add the necessary properties.



Option 2: via asadmin CLI

use of the asadmin list/get/set for HTTP properties

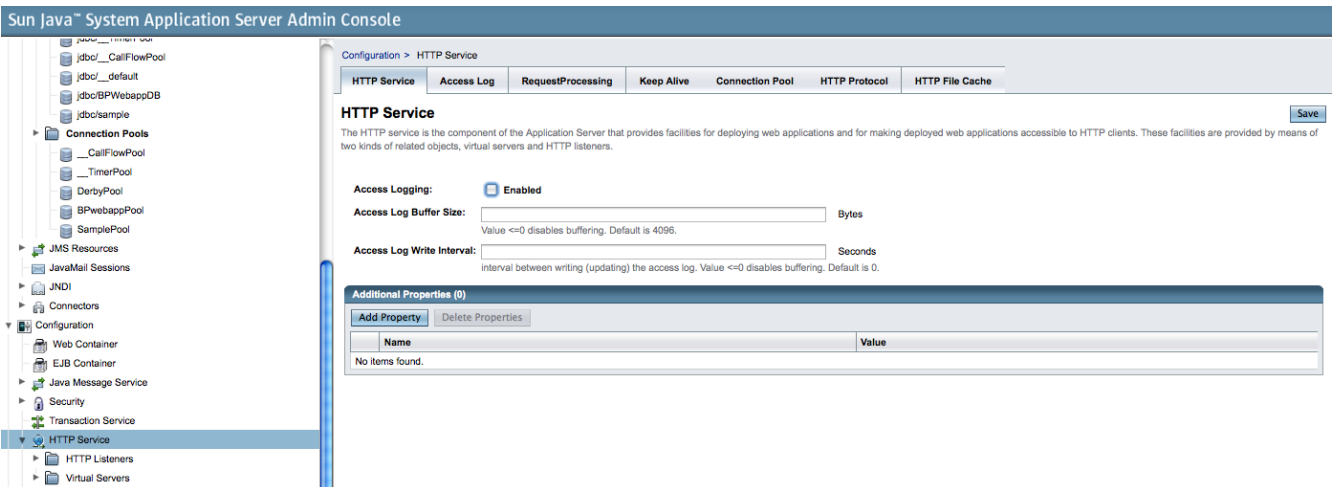
Example:

```
asadmin set server.resources.jdbc-connection-
pool.<mypoolname>.property.maxStatements=200
```

### Tip 11 – Disable Access Logging

Option 1 - via Administration console:

1. Login at the administration URL: `http://<yourhostname>:4848`
2. Expand the node on the left hand side by clicking on Configuration --> HTTP Service.
3. Click on HTTP Service on right hand side.



4. Uncheck the box for “Access Logging”.

Option 2: via asadmin CLI

use of the asadmin list/get/set for HTTP properties

Example:

```
asadmin get server.http-
service.property.accessLoggingEnabled

asadmin set server.http-
service.property.accessLoggingEnabled = false
```

### Tip for CoolThreads UltraSPARC T1/T2 Tuning

### Rotating access log

Separate the access logs, transaction logs and imq files onto as many disks as possible. If you have two disks, put the XA transaction logs (instance\_directory/logs/tx) on one disk and the imq files (instance\_directory/imq/instances/imqbroker/fs350) on another. If you have more disks, put the imqbroker txn file on a separate disk.

If you run multiple instances of the application server, move the logs for each instance onto separate disks as much as possible.

Mount the disk with these options: nologging,directio,noatime.

If the disks become a bottleneck, consider using an external disk with write cache.

### Use libumem

To configure the GlassFish application server, see below:

For sh:

```
LD_PRELOAD=/usr/lib/libumem.so asadmin start-domain  
[arguments]
```

For csh:

```
env LD_PRELOAD=/usr/lib/libumem.so asadmin start-domain  
[arguments]
```

### Use of FX scheduling class

On systems with UltraSPARC T1 processors, it is recommended to put the application server into the FX scheduling class. This standard feature of the Solaris OS is relevant on this platform because the default time quanta for scheduling doesn't match with the time quanta for processing a web request. By putting the application server in FX class, a simple web request is more likely to be processed within a single dispatch as it will not be preempted by a different thread, causing an increase in throughput.

1.Find the process ID of the GlassFish process:

```
ps -ef | grep appserv
```

2.Find the process ID of the MQ Broker daemon:

```
ps -ef | grep mqbrokerd
```

3.For each process ID, execute:

```
/usr/bin/priocntl -s -c FX -m 59 -p 59 -i <process id of  
appserver>
```

### Network Configuration

If you are using more than one network interface, then you will need to make sure that all the network interrupts are not going to the same core. Use the following script to enable interrupts on one strand and disable interrupts on the remaining three strands of a core. You can verify the processor state change with the psinfo command.

Ex : psinfo output before running the script

```
# psrinfo | more
0      on-line   since 01/10/2008 15:06:01
1      on-line   since 01/13/2008 13:50:00
2      on-line   since 01/13/2008 13:50:00
3      on-line   since 01/13/2008 13:50:00
.....
psrinfo output after running the script
0      on-line   since 01/10/2008 15:06:01
1      no-intr   since 01/13/2008 13:55:16
2      no-intr   since 01/13/2008 13:55:16
3      no-intr   since 01/13/2008 13:55:16
```

Table 6. Network interrupt script:

```
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
```

---