Case study: Tuning WebSphere Application Server V7 and V8 for performance

Updated for WebSphere Application Server V8

IBM® WebSphere® Application Server supports an ever-growing range of applications, each with their own unique set of features, requirements, and services. Just as no two applications will use an application server in exactly the same way, no single set of tuning parameters will likely provide the best performance for any two different applications. Most applications will generally realize some performance improvement from tuning in three core areas: the JVM, thread pools, and connection pools. This article uses the Apache DayTrader Performance Benchmark Sample application to demonstrate what you can tune and how to go about tuning it, depending on the major server components and resources that your application uses. Updated for WebSphere Application Server V8. This content is part of the IBM WebSphere Developer Technical Journal.

David Hare an Advisory Software Engineer with the WebSphere Application Server Performance and Benchmarking organization in Research Triangle Park, North Carolina. His primary focus has been on the DayTrader performance benchmark, performance tuning, and the brand new Liberty Profile.

Christopher Blythe was an Advisory Software Engineer and technical team lead in the WebSphere Application Server Performance and Benchmarking organization at the time of original publication.

22 June 2011 (First published 30 September 2009)

Also available in Chinese, Japanese, Portuguese

Introduction

IBM WebSphere Application Server is a robust, enterprise class application server that provides a core set of components, resources, and services that developers can utilize in their applications. Every application has a unique set of requirements and often uses an application server’s resources in vastly different ways. In order to provide a high degree of flexibility and support for this wide variety of applications, WebSphere Application Server offers an extensive list of tuning "knobs" and parameters that you can use to enhance an application’s performance.

Default values for the most commonly used tuning parameters in the application server are set to ensure adequate performance out of the box for the broadest range of applications. However, because no two applications are alike or use the application server in exactly the same fashion, there is no guarantee that any single set of tuning parameters will be perfectly suited for every application. This reality highlights how important it is for you to conduct focused performance testing and tuning against your applications.

This article discusses several of the most commonly used parameters in WebSphere Application Server V7 and V8 (and previous releases) and the methodologies used to tune them. Unlike many documented tuning recommendations, this article uses the Apache DayTrader Performance Benchmark Sample application as a case study to provide context to the discussion. With the DayTrader application, you will be able to clearly identify the key server components in use, perform focused tuning in these areas, and witness the benefit associated with each tuning change.

The information in this article applies to both WebSphere Application Server Versions 7 and 8, except where noted.

Before proceeding, here are a few additional things to keep in mind when tuning your application server for performance:

- Increased performance can often involve sacrificing a certain level of feature or function in the
application or the application server. The tradeoff between performance and feature must be weighed carefully when evaluating performance tuning changes.

Several factors beyond the application server can impact performance, including hardware and OS configuration, other processes running on the system, performance of back-end database resources, network latency, and so on. You must factor in these items when conducting your own performance evaluations.

The performance improvements detailed here are specific to the DayTrader application, workload mix, and the supporting hardware and software stack described here. Any performance gains to your application that result from the tuning changes detailed in this article will certainly vary, and should be evaluated with your own performance tests.

The DayTrader application

The Apache DayTrader Performance Benchmark Sample application simulates a simple stock trading application that lets users login/logout, view their portfolio, look up stock quotes, buy and sell stock share, and manage account information. DayTrader not only serves as an excellent application for functional testing, but it also provides a standard set of workloads for characterizing and measuring application server and component level performance. DayTrader (and the Trade Performance Benchmark Sample application by IBM on which DayTrader was originally based) was not written to provide optimal performance. Rather, the applications is intended to conduct relative performance comparisons between application server releases and alternative implementation styles and patterns.

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

In order for you to evaluate a few common Java EE persistence and transaction management patterns, DayTrader provides three different implementations of the business services. These implementations (or run time modes) are shown in Table 1.

<table>
<thead>
<tr>
<th>Run time mode</th>
<th>Pattern Description</th>
</tr>
</thead>
</table>
| Direct        | Servlet to JDBC     | Create, read, update, and delete (CRUD) operations are performed directly against the database using custom JDBC code. Database connections, commits,
These run time modes are included within the scope of this article to demonstrate the impact each tuning change has on these three common Java EE persistence and transaction implementation styles.

Navigating the WebSphere Application Server tunables
As mentioned earlier, the importance of understanding the application architecture, the server components, and the resources utilized by the application cannot be stressed enough when conducting performance tuning. With this knowledge, you can quickly filter the list of tunable parameters and focus on a core set of tunables that directly impact your application.

Performance tuning generally starts with the Java Virtual Machine (JVM), which serves as the foundation for the application server. From that point forward, tuning is primarily driven by the application server components that are used by the application. For example, you can identify some of the key tunable server components of the DayTrader application using the architecture diagram (Figure 1):

- Web and EJB containers
- Associated thread pools
- Database connection pools
- Default messaging provider.

The remainder of this article discusses details of several tuning options that impact the performance of DayTrader based on the components listed above. These options have been divided into these sections:

**Basic tuning**: This group of tuning parameters covers a few of the most commonly tuned and heavily used application server components, starting with the JVM. These settings traditionally provide the most bang for the buck.

**Advanced tuning**: This secondary group of advanced tuning parameters are generally specific to a certain scenario and are often used to squeak the absolute most performance out of a system.

**Asynchronous messaging tuning**: The options discussed here are specific to applications that utilize WebSphere Application Server’s messaging components for asynchronous messaging.

The tuning parameters grouped in these sections are presented with a detailed discussion of applicability, and information related to functional tradeoffs and their ultimate impact on performance (for each of the persistence and transaction management patterns, if appropriate). Tools that can aid in the tuning process for a given parameter are also presented. Links to associated WebSphere Application Server Information Center documentation or other related resources are provided in each section, and also summarized in the Resources section.

**Basic tuning**
Discussed in this section:

a. JVM heap size
b. Thread pool size
c. Connection pool size
Data source statement cache sized.

ORB pass by reference.
a. JVM heap size

JVM heap size parameters directly influence garbage collection behavior. Increasing the JVM heap size permits more objects to be created before an allocation failure occurs and triggers a garbage collection. This naturally enables the application to run for a longer period of time between garbage collection (GC) cycles. Unfortunately, an associated downside to increased heap size is a corresponding increase in the amount of time needed to find and process objects that should be garbage collected. Consequently, JVM heap size tuning often involves a balance between the interval between garbage collections and the pause time needed to perform the garbage collection.

In order to tune the JVM heap size, verbose GC needs to be enabled. This can be done in the WebSphere Application Server administrative console by navigating to **Servers => Application servers => server_name => Process definition => Java Virtual Machine**. By enabling verbose GC, the JVM prints out useful information at each garbage collection, such as the amount of free and used bytes in the heap, the interval between garbage collections, and the pause time. This information is logged to the native_stderr.log file, which can then be used by various tools to visualize the heap usage.

WebSphere Application Server's default heap settings are 50 MB for the initial heap size and 256 MB for the maximum heap size. It is generally recommended to set the minimum and maximum heap sizes to the same value for maximum performance. See below to understand why.

**Testing**

Four lab tests were conducted with verbose GC enabled, with the initial and maximum heap sizes both set to 256 MB, 512 MB, 1024 MB, and 2048 MB, respectively. You can analyze verbose GC manually, but graphical tools like the **IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer** tool (packaged with the IBM Support Assistant, which is freely available for download) can make the process considerably easier. That tool was used in the lab tests to view the verbose GC data so that adjustments could be made in finding the appropriate heap size for the DayTrader application.

One of the first items to monitor in the verbose garbage collection output is the free heap after collection. This metric is commonly used to determine if any Java memory leaks are present within the application. If this metric does not reach a steady state value and continues to decrease over time, then you have a clear indication that a memory leak is present within the application. The free heap after collection metric can also be used in conjunction with the heap size to calculate the working set of memory (or “footprint”) being used by the server and application. Simply subtract the free heap value from the total heap size to get this value.

**Figure 2. Free heap after collection summary**
Be aware that while the verbose GC data and the chart in Figure 2 can help detect memory leaks within the Java heap, it cannot detect a native memory leak. Native memory leaks occur when native components (like vendor native libraries written in C/C++ and called via the Java Native Interface (JNI) APIs) leak native memory outside the Java heap. In these situations, it is necessary to use platform specific tools (like the `ps` and `top` commands on Linux® platforms, Perfmon on Windows® platforms, and so on) to monitor the native memory usage for the Java processes. The WebSphere Application Server Support page has detailed documentation on diagnosing memory leaks.

Two additional metrics to key in on are the garbage collection intervals and the average pause times for each collection. The GC interval is the amount of time in between garbage collection cycles. The pause time is the amount of time that a garbage collection cycle took to complete. Figure 3 charts the garbage collection intervals for each of the four heap sizes. They averaged out to 0.6 seconds (256 MB), 1.5 seconds (512 MB), 3.2 seconds (1024 MB), and 6.7 seconds (2048 MB).

Figure 3. Garbage collection intervals
Figure 4 shows the **average pause times** for each of the four heap sizes. In lab testing, they averaged out to 62 ms (256 MB), 69 ms (512 MB), 83 ms (1024 MB), and 117 ms (2048 MB). This clearly demonstrates the standard tradeoff associated with increasing the Java heap size. As heap size increases, the interval between GCs increase, enabling more work to be performed before the JVM pauses to execute its garbage collection routines. However, increasing the heap also means that the garbage collector must process more objects and, in turn, drives the GC pause times higher.

The GC intervals and pause times together make up the amount of time that was spent in garbage collection.
The percentage of time spent in garbage collection is shown in the IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer tool and can be calculated using this formula:

\[
\text{% Time in GC} = \frac{\text{Avg Pause Time}}{\text{GC Interval + Avg Pause Time}}
\]

The amount of time spent in garbage collection and the resulting throughput (requests per second) for each of these runs is displayed in Figure 5.

**Figure 5. Time spent in garbage collection and throughput**

![Graph showing time spent in garbage collection and throughput]

A recommended setting for initial and maximum heap size for the DayTrader application would therefore be 1024 MB. At this point, you reach a point of diminishing marginal returns where increasing the heap size further does not yield a proportional performance benefit. This provides a good balance of higher garbage collection intervals and lower pause times, which results in a small amount of time spent in garbage collection.

Another important aspect of JVM tuning is the **garbage collection policy**. The three main GC policies are:

- **optthruput**: (default in V7) Performs the mark and sweep operations during garbage collection when the application is paused to maximize application throughput.

- **optavpgpause**: Performs the mark and sweep concurrently while the application is running to minimize pause times; this provides the best application response times.

- **gencon**: (default in V8) Treats short-lived and long-lived objects differently to provide a combination of lower pause times and high application throughput.

The DayTrader application does not use many long-lived objects and typically runs best with the default optthruput GC policy. However, each application is different and so you should evaluate each GC policy to find the best fit for your application. The developerWorks article [Garbage collection policies](http://www.ibm.com/developerworks/websphere/techjournal/0909_bly...) is a great resource for learning more about the GC policies.

Figure 6 presents the performance gain achieved (determined from the application throughput) by tuning the JVM heap size for each of the different DayTrader run time modes. In the chart, the blue bars always represent the baseline throughput values and the red bars represent the throughput values after adjusting the discussed tuning parameter. In order to present the relative throughput differences between the various run time modes, all measurements are compared against the EJB mode baseline.

For example, prior to tuning, Session Direct and Direct mode are 23% and 86% faster than the EJB mode baseline, respectively. The line on the secondary axis represents the overall performance improvement for each of the run time modes. In this case, the JVM tuning resulted in varying improvements for the
three run time modes due to the unique object allocation patterns associated with each. These improvements ranged from only 3% (for JDBC mode) up to 9% (for EJB mode).

This information will be shown at the end of each tuning parameter, where applicable. The performance improvement after the application of each tuning parameter is cumulative, building upon the parameters from the previous sections. A chart at the end of this article (Figure 22) exhibits the overall performance improvement achieved with all tuning changes applied.

**Figure 6. Performance benefit of tuned JVM heap size**

![JVM heap size graph](image)

**JVM minimum heap size = Maximum heap size**

As mentioned earlier, it is generally recommended that the minimum and maximum heap sizes are set to the same value to maximize run time performance. This prevents the JVM from compacting, which dynamically resizes the heap. This is a very expensive operation and can be as much as 50% of garbage collection pause times. Setting the min heap size equal to the max heap size also makes garbage collection analysis easier by fixing a key parameter to a constant value. The tradeoff for setting the minimum and maximum heap sizes to the same value is that the initial startup of the JVM will be slower because the JVM must allocate the larger heap.

There are scenarios where setting the minimum and maximum heap settings to different values can be advantageous. One such scenario is when multiple application server instances running different workloads are hosted on the same server. In this scenario, the JVM can dynamically respond to the changing workload requirements and more effectively utilize system memory.

To illustrate the benefit of setting the minimum and maximum heap sizes to the same value, a test was done where the max heap size was fixed at 1024 MB (1 GB) and the performance was measured as the min heap size was increased. The results are charted in Figure 7 below.

**Figure 7. Performance benefit of setting min=max**

![Heap size comparison graph](image)
The baseline is the default out of the box settings of 50 MB min and 256 MB max. For this workload, it is apparent that simply specifying a max heap size of 1 GB does not provide much improvement -- less than 2% -- because the amount of compaction is so great. However, as the minimum heap size is increased to closer match the maximum heap size, the amount of compaction is decreased, which reduces the amount of time spent in garbage collection. Therefore the overall application throughput increases by almost 10%, which closely matches the percentage of time spent in garbage collection in the default configuration.

**Tuning the nursery size**

In WebSphere Application Server V8, the default garbage collection policy has changed to gencon, which separates the heap into two spaces, a nursery space and a tenured space. The nursery space is where the short-lived objects reside and, correspondingly, the tenured space is where the long-lived objects reside. Objects that stick around are moved from the nursery space to the tenured space after a certain period of time. This is a beneficial GC policy because nursery collections have a much smaller pause time than global collections.

The important tuning setting here is the nursery size (-Xmn). The default value is 25% of the maximum heap size, which means the tenured space is 75%. For an application with mostly longer lived objects this is a pretty good value. However, for applications with more short-lived objects, a performance improvement can be seen by increasing the nursery size.

A transaction-based application like DayTrader is an example of an application that will see a benefit from a larger nursery size, because many objects are created to perform the transaction and then thrown away once the transaction has completed. To illustrate this, a test was done with the previous settings of 1024 MB min/max heap, with a nursery size of 256 MB (default), 512 MB, and 768 MB. The results are shown in Figure 8.

**Figure 8. Performance benefit of tuning the nursery size**

For more information:

- Tuning the IBM virtual machine for Java
- Java technology, IBM style: Garbage collection policies, Part 1
- IBM Support Assistant

**b. Thread pool size**

Each task performed by the server runs on a thread obtained from one of WebSphere Application Server’s many thread pools. A thread pool enables components of the server to reuse threads, eliminating the need to create new threads at run time to service each new request. Three of the most
commonly used (and tuned) thread pools within the application server are:

**Web container:** Used when requests come in over HTTP. In the DayTrader architecture diagram (Figure 1), you can see that most traffic comes into DayTrader through the Web Container thread pool.

**Default:** Used when requests come in for a message driven bean or if a particular transport chain has not been defined to a specific thread pool.

**ORB:** Used when remote requests come in over RMI/IIOP for an enterprise bean from an EJB application client, remote EJB interface, or another application server.

The important tunable options associated with thread pools are shown in Table 2.

**Table 2. Tunable thread pool options**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum size</td>
<td>The minimum number of threads permitted in the pool. When an application server starts, no threads are initially assigned to the thread pool. Threads are added to the thread pool as the workload assigned to the application server requires them, until the number of threads in the pool equals the number specified in the minimum size field. After this point in time, additional threads are added and removed as the workload changes. However, the number of threads in the pool never decreases below the number specified in the minimum size field, even if some of the threads are idle.</td>
</tr>
<tr>
<td>Maximum size</td>
<td>Specifies the maximum number of threads to maintain in the default thread pool.</td>
</tr>
<tr>
<td>Inactivity timeout</td>
<td>Specifies the amount of inactivity (in milliseconds) that should elapse before a thread is reclaimed. A value of 0 indicates not to wait, and a negative value (less than 0) means to wait forever.</td>
</tr>
</tbody>
</table>

Assuming that the machine contains a single application server instance, a good practice is to use 5 threads per server CPU core for the default thread pool, and 10 threads per server CPU for the ORB and Web container thread pools. For a machine with up to 4 CPUs, the default settings are usually a good start for most applications. If the machine has multiple application server instances, then these sizes should be reduced accordingly. Conversely, there could be situations where the thread pool size might need to be increased to account for slow I/O or long running back-end connections. Table 3 shows the default thread pool sizes and inactivity timeouts for the most commonly tuned thread pools.

**Table 3. Default thread pool sizes and inactivity timeouts**

<table>
<thead>
<tr>
<th>Thread pool</th>
<th>Minimum size</th>
<th>Maximum size</th>
<th>Inactivity timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>20</td>
<td>20</td>
<td>5000 ms</td>
</tr>
<tr>
<td>ORB</td>
<td>10</td>
<td>50</td>
<td>3500 ms</td>
</tr>
<tr>
<td>Web container</td>
<td>50</td>
<td>50</td>
<td>60000 ms</td>
</tr>
</tbody>
</table>

Thread pool settings can be changed in the administrative console by navigating to **Servers => Application Servers => server_name => Thread Pool**. You can also use the **Performance Advisors** to get recommendations on thread pool sizes and other settings.

The IBM Tivoli® Performance Viewer is a tool embedded in the administrative console that lets you view the PMI (Performance Monitoring Infrastructure) data associated with almost any server component. The viewer provides advice to help tune systems for optimal performance and recommends alternatives to inefficient settings. See the **WebSphere Application Server Information Center** for instructions on enabling and viewing PMI data with the Tivoli Performance Viewer.

Figure 9 shows the PMI data for the Web container thread pool while the DayTrader application was ramped up and run under steady-state, peak load. The pool size (orange) is the average number of threads in the pool, and the active count (red) is the number of concurrently active threads. This graph shows that the default setting of 50 maximum Web container threads works well in this case, as all 50 threads have not been allocated and the average concurrent workload is using around 18 threads. Since the default thread pool sizing was sufficient, no modifications were made to the thread pool sizing.

**Figure 9. PMI data for Web container thread pool**
Prior to WebSphere Application Server V6.x, a one-to-one mapping existed between the number of concurrent client connections and the threads in the Web container thread pool. In other words, if 40 clients were accessing an application, 40 threads were needed to service the requests. In WebSphere Application Server V6.0 and 6.1, Native IO (NIO) and Asynchronous IO (AIO) were introduced, providing the ability to scale to thousands of client connections using a relatively small number of threads. This explains why, in Figure 9, an average of 18 threads were used to service 50 concurrent client connections from the HTTP load driver. Based on this information, the thread pool size could be lowered to reduce the overhead involved with managing a larger than needed thread pool. However, this would lessen the ability of the server to respond to load spikes in which a large number of threads were actually needed. Careful consideration should be taken in determining the thread pool sizes, including what the expected average and peak workload could potentially be.

c. Connection pool size

Each time an application attempts to access a back-end store (such as a database), it requires resources to create, maintain, and release a connection to that data store. To mitigate the strain that this process can place on overall application resources, the application server enables you to establish a pool of back-end connections that applications can share on an application server. Connection pooling spreads the connection overhead across several user requests, thereby conserving application resources for future requests. The important tunable options associated with connection pools are shown in Table 4.

<table>
<thead>
<tr>
<th>Table 4. Connection pool tuning options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting</td>
</tr>
<tr>
<td>Minimum connections</td>
</tr>
<tr>
<td>Maximum connections</td>
</tr>
<tr>
<td>Thread inactivity timeout</td>
</tr>
</tbody>
</table>

The goal of tuning the connection pool is to ensure that each thread that needs a connection to the database has one, and that requests are not queued up waiting to access the database. For the
DayTrader application, each task performs a query against the database. Since each thread performs a task, each concurrent thread needs a database connection. Typically, all requests come in over HTTP and are executed on a Web container thread. Therefore, the maximum connection pool size should be at least as large as the maximum size of the Web container thread pool.

Be aware, though, this is not a best practice for all scenarios. Using a connection pool as large or larger than the Web container thread pool ensures that there are no threads waiting for a connection and provides the maximum performance for a single server. However, for an environment that has numerous application servers all connecting to the same back end database, careful consideration should be placed on the connection pool size. If, for example, there are ten application servers all accessing a database with 50 connections in the connection pool, that means up to 500 connections could be requested at a time on the database. This type of load could easily cause problems on the database.

A better approach here is to use the “funneling” method, in which the number of Web container threads is larger than the number of connections in the connection pool. This ensures that under extreme load, not all active threads are getting a connection to the database at a single moment. This will produce longer response times, but will make the environment much more stable.

Overall, the general best practice is to determine which thread pools service tasks that require a DataSource connection and to size the pools accordingly. In this case, since we are focused on achieving the best performance for a single server, the maximum connection pool size was set to the sum of the maximum size of the default and Web container thread pools (70). The connection pool settings can be changed in the administrative console by navigating to Resources => JDBC => Data Sources => data_source => Connection pool properties. Bear in mind that all applications might not be as well behaved as DayTrader from a connection management perspective, and therefore might use more than one connection per thread.

Figure 10 shows the PMI data for the connection pool while the DayTrader application was running under steady state peak load with the default connection pool sizes of 1 minimum/10 maximum. FreePoolSize (orange) is the number of free connections in the pool and UseTime (green) is the average time (in ms) that a connection is used. This graph shows that all 10 connections were always in use. In addition to the charted metrics, the table also shows some other key metrics: WaitingThreadCount shows 33 threads waiting for a connection to the database with an average WaitTime of 8.25 ms, and the pool overall is 100% occupied, as shown by the PercentUsed metric.

**Figure 10. PMI metrics before tuning the connection pool**
Figure 11 shows the same chart after tuning the connection pool to a size of 10 minimum/70 maximum. This shows that there were plenty of free connections available and no threads were waiting for a connection, which produces much faster response times.

Figure 11. PMI metrics after tuning the connection pool

Figure 12. Performance benefit of tuned connection pool sizes

Case study: Tuning WebSphere Application Server V7 and V8 for per...
For more information:

- Tuning application servers
- Data access tuning parameters
- Why you want to use the performance advisors

d. Data source statement cache size

Data source statement cache size specifies the number of prepared JDBC statements that can be cached per connection. The WebSphere Application Server data source optimizes the processing of prepared statements and callable statements by caching those statements that are not being used in an active connection. If your application utilizes many statements like DayTrader does, then increasing this parameter can improve application performance. The statement cache size can be configured by navigating to Resources => JDBC => Data sources => data_source => WebSphere Application Server data source properties.

The data source statement cache size can be tuned using a few different methods. One technique is to review the application code (or an SQL trace gathered from the database or database driver) for all unique prepared statements, and ensure the cache size is larger than that value. The other option is to iteratively increase the cache size and run the application under peak steady state load until the PMI metrics report no more cache discards. Figure 13 shows the same PMI chart of the connection pool, this time with the data source statement cache size increased from the default size (which is 10) to 60. The metric PrepStmtCacheDiscardCount (red) is the number of statements that are discarded because the cache is full. Looking back at the chart in Figure 11, before tuning the data source statement cache size, the number of statements discarded was over 1.7 million. The chart in Figure 13 shows there were no statement discards after tuning the cache size.

Figure 13. PMI metrics after tuning the data source statement cache size

Figure 14. Performance benefit of increased data source statement cache size
For more information:

Data access tuning parameters

e. ORB pass by reference

The Object Request Broker (ORB) pass by reference option determines if pass by reference or pass by value semantics should be used when handling parameter objects involved in an EJB request. This option can be found in the administrative console by navigating to **Servers => Application Servers => server_name => Object Request Broker (ORB)**. By default, this option is disabled and a copy of each parameter object is made and passed to the invoked EJB method. This is considerably more expensive than passing a simple reference to the existing parameter object.

To summarize, the ORB pass by reference option basically treats the invoked EJB method as a local call (even for EJBs with remote interfaces) and avoids the requisite object copy. If remote interfaces are not absolutely necessary, a slightly simpler alternative which does not require tuning is to use EJBs with local interfaces. However, by using local instead of remote interfaces, you lose the benefits commonly associated with remote interfaces, location transparency in distributed environments, and workload management capabilities.

The ORB pass by reference option will only provide a benefit when the EJB client (that is, servlet) and invoked EJB module are located within the same classloader. This requirement means that both the EJB client and EJB module must be deployed in the same EAR file and running on the same application server instance. If the EJB client and EJB modules are mapped to different application server instances (often referred to as split-tier), then the EJB modules must be invoked remotely using pass by value semantics.

Because the DayTrader application contains both WEB and EJB modules in the same EAR, and both are deployed to the same application server instance, the ORB pass by reference option can be used to realize a performance gain. As indicated by the measurements shown in Figure 15, this option is extremely beneficial for DayTrader where all requests from the servlets are passed to the stateless session beans over a remote interface -- except in direct mode, where the EJB container is bypassed in lieu of direct JDBC and manual transactions.

**Figure 15. Performance benefit of ORB pass by reference**
Advanced tuning

Discussed in this section:

a. **Servlet caching**
b. **HTTP transport persistent connections**
c. **Large page support**
d. **Disabling unused services**
e. **Web server location**

**a. Servlet caching**

WebSphere Application Server's DynaCache provides a general in-memory caching service for objects and page fragments generated by the server. The DistributedMap and DistributedObjectCache interfaces can be used within an application to cache and share Java objects by storing references to these objects in the cache for later use. Servlet caching, on the other hand, enables servlet and JSP response fragments to be stored and managed by a customizable set of caching rules.

In the DayTrader application, a Market Summary is displayed on every access of a user’s home page. This summary contains a list of the top five gaining and losing stocks, as well as the current stock index and trading volume. This activity requires the execution of several database lookups and therefore significantly delays the loading of the user's home page. With servlet caching, the marketSummary.jsp can be cached, virtually eliminating these expensive database queries to improve the response time for the user home page. The refresh interval for the cached object can be configured, and is set to 60 seconds in the example shown in Listing 1. Dynacache could also be used to cache other servlet/JSP fragments and data within DayTrader. This example demonstrates the improvement you can achieve through caching to avoid complex server operations.

Servlet caching can be enabled in the administrative console by navigating to **Servers => Application servers => server_name => Web container settings => Web container**. The URI path to the servlet or JSP to be cached must be defined in a cachespec.xml file, which is placed inside the Web module's WEB-INF directory. For the marketSummary.jsp in DayTrader, the cachespec.xml looks similar to Listing 1.

**Listing 1. cachespec.xml**
HTTP transport persistent connections

Persistent connections specify that an outgoing HTTP response should use a persistent (keep-alive) connection instead of a connection that closes after one request or response exchange occurs. In many cases, a performance boost can be achieved by increasing the maximum number of persistent requests that are permitted on a single HTTP connection. SSL connections can see a significant performance gain by enabling unlimited persistent requests per connection because SSL connections incur the costly overhead of exchanging keys and negotiating protocols to complete the SSL handshake process. Maximizing the number of requests that can be handled per connection minimizes the impact of this overhead. Also, high throughput applications with fast response times can realize a performance gain by keeping the connections open, rather than building up and closing the connection on each request. When this property is set to 0 (zero), the connection stays open as long as the application server is running. However, if security is a concern, then careful consideration should be placed on this setting, as this parameter can help prevent denial of service attacks when a client tries to hold on to a keep-alive connection.

HTTP transport persistent connections settings can be set in the administrative console by navigating to Servers => Application servers => server_name => Ports. Once there, click on View associated transports for the port associated with the HTTP transport channel settings you want to change. During DayTrader testing, the Maximum persistent requests per connection value was changed from 100 (the default) to unlimited. The charts in Figure 17 show the throughput results of accessing a simple "Hello World" servlet over standard HTTP (non-SSL) and HTTPS (SSL), both before and after enabling unlimited persistent requests per connection.

Figure 16. Performance benefit of servlet caching

http://www.ibm.com/developerworks/websphere/techjournal/0909_bly...
c. Large page support

Several platforms provide the ability to establish a large contiguous section of memory using memory pages that are larger than the default memory page size. Depending on the platform, large memory page sizes can range from 4 MB (Windows) to 16 MB (AIX) versus the default page size of 4KB. Many applications (including Java-based applications) often benefit from large pages due to a reduction in CPU overhead associated with managing smaller numbers of large pages.

In order to use large memory pages, you must first define and enable them within the operating system. Each platform has different system requirements that must be configured before large page support can be enabled. The WebSphere Application Server Information Center documents each of these steps by platform:

Tuning AIX® systems
Tuning Linux systems
Tuning Solaris™ systems
Tuning Windows systems

Once configured within the operation system, large page support within the JVM can be enabled by specifying `-Xlp` in the Generic JVM Arguments settings in the administrative console at Servers => Application servers => server_name => Process definition => Java Virtual Machine. Be aware that if large pages are enabled, the operating system will set aside a large continuous chunk of memory for use by the JVM. If the amount of memory remaining is not enough to handle the other applications that are running, paging (swapping pages in memory for pages on the hard disk) could occur, which will significantly reduce system performance.

Figure 18. Performance benefit of large pages

For more information:

HTTP transport channel settings

For more information:

Case study: Tuning WebSphere Application Server V7 and V8 for per... http://www.ibm.com/developerworks/websphere/techjournal/0909_bly...
d. Disabling unused services

Disabling unused services that an application doesn't need can improve performance. One such example is PMI. It is important to note that PMI must be enabled to view the metrics documented earlier in this article and to receive advice from the performance advisors. While disabling PMI removes the ability to see this information, it also provides a small performance gain. PMI can be disabled on an individual application server basis in the administrative console by navigating to Monitoring and Tuning => Performance Monitoring Infrastructure (PMI).

![Figure 19. Performance benefit of disabling PMI](image)

For more information:
- Enabling PMI data collection

e. Web server location

Web servers like IBM HTTP Server are often used in front of WebSphere Application Server deployments to handle static content or to provide workload management (WLM) capabilities. In versions of the WebSphere Application Server prior to V6, Web servers were also needed to effectively handle thousands of incoming client connections, due to the one-to-one mapping between client connections and Web container threads (discussed earlier). In WebSphere Application Server V6 and later, this is no longer required with the introduction of NIO and AIO. For environments that use Web servers, the Web server instances should be placed on dedicated systems separate from the WebSphere Application Server instances. If a Web server is collocated on a system with a WebSphere Application Server instance, they will effectively share valuable processor resources, reducing overall throughput for the configuration.

A DayTrader test was conducted with IBM HTTP Server placed locally on the same machine as WebSphere Application Server, and a second test was conducted with the Web server located remotely on a separate dedicated machine. Table 5 shows the percentage of CPU cycles consumed by each process when the Web server and application server were collocated on the same system. As you can see from the results, approximately 25% of the CPU was consumed by the HTTP Server process that corresponds to a single CPU in the four-CPU system that was used for this test.

<table>
<thead>
<tr>
<th>Table 5. CPU utilization with HTTP Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
</tr>
<tr>
<td>WebSphere Application Server</td>
</tr>
<tr>
<td>IBM HTTP Server</td>
</tr>
</tbody>
</table>

Figure 20 shows the throughput and response time comparison of these two scenarios, as well as the case where there was no Web server at all.
Asynchronous messaging tuning

The majority of this paper has focused so far on the core Web serving and persistence aspects of the DayTrader application and WebSphere Application Server. We will now shift focus to how DayTrader uses JMS components to perform asynchronous processing of buy/sell orders and to monitor price quote changes. The DayTrader benchmark application contains two messaging features that can be independently enabled or disabled:

- **Async order processing**: Asynchronous buy and sell order processing is handled by a JMS queue and an MDB.
- **Quote price consistency tracking**: A JMS topic and MDB are used to monitor quote price changes associated with stock buy and sell orders.

There are two primary tuning options associated with the messaging configuration that will have a significant impact on performance: the message store type, and the message reliability. Along with these, a more advanced tuning technique that will yield additional performance gains is placing the transaction logs and file store (if applicable) on a fast disk. Each of these topics and their corresponding performance gains are discussed in detail below.

Discussed in this section:

a. **Message store type**
b. **Message reliability levels**
c. **Moving transaction log and file store to a fast disk**

**a. Message store type**

WebSphere Application Server's internal messaging provider maintains the concept of a messaging engine "data store." The data store serves as the persistent repository for messages handled by the engine. When a messaging engine is created in a single-server environment, a file-based store is created to use as the default data store. In WebSphere Application Server V6.0.x, the default data store was provided by a local, in-process Derby database. The file and Derby-based data stores are convenient for in single-server scenarios, but do not provide the highest level of performance, scalability, manageability, or high availability. To meet these requirements, you can use a remote database data store:

- **Local Derby database data store**: With this option, a local, in-process Derby database is used to store the operational information and messages associated with the messaging engine. Although convenient for development purposes, this configuration used valuable cycles and memory within the application server to manage the stored messages.
- **File-based data store**: (default) If the message engine is configured to use a file-based data store,
operating information and messages are persisted to the file system instead of a database. This performs faster than the local Derby database and, when a fast disk such as a redundant array of independent disks (RAID) is used, can perform just as fast as a remote database. The test results shown below did not use a RAID device for the file-based data store and do not reflect this additional improvement.

Remote database data store: In this configuration, a database residing on a remote system is configured to act as the message engine data store. This frees up cycles for the application server JVM process that were previously used to manage the Derby database or file-based stores, enabling a more performant, production level database server (such as IBM DB2® Enterprise Server) to be used. One technical advantage of using a database for the data store is that some J2EE™ applications can share JDBC connections to benefit from one-phase commit optimization. For more information see information on sharing connections to benefit from one-phase commit optimization. File store does not support this optimization.

DayTrader was run in EJB asynchronous mode with these three different message store types. During these runs, the trace specification org.apache.geronimo.samples.daytrader.util.Log=all was enabled to capture the time to receive messages for the TradeBrokerMDB. When measuring asynchronous messaging performance, it is always important to base the measurements on the MDB response times that are asynchronous -- and not the actual page response times, which are synchronous. The results in Figure 21 show that a remote database data store yields the best performance, as it provides the fastest MDB response time and the highest throughput.

Figure 21. Performance comparison of message store types

For more information:

Relative advantages of a file store and a data store

b. Message reliability levels
Message reliability is an important component for any messaging system and the WebSphere Application Server messaging provider provides five different levels of reliability:

Best effort non-persistent
Express non-persistent
Reliable non-persistent
Reliable persistent
Assured persistent.

Persistent messages are always stored to some form of persistent data store, while non-persistent messages are generally stored in volatile memory. There is a trade-off here between reliability of
message delivery and the speed with which messages are delivered. Figure 22 shows the results for two test runs using a file store with different reliability levels: assured persistent and express non-persistent. The results clearly illustrate that as the reliability level decreases, the faster the messages can be processed.

Figure 22. Performance comparison of message reliability levels

For more information:

Message reliability levels - JMS delivery mode and service integration quality of service

c. Moving transaction log and file store to a fast disk

Since disk I/O operations are costly, storing log files on fast disks such as a RAID can greatly improve performance. In most RAID configurations, the task of writing data to the physical media is shared across the multiple drives. This technique yields more concurrent access to storage for persisting transaction information, and faster access to that data from the logs.

The Transaction log directory can be set in the administrative console by navigating to Servers => Application Servers => server_name => Container Services => Transaction Service.

The File store log directory can be specified during the creation of an SIBus member using the -logDirectory option in the AdminTask addSIBusMember command or via the administration console SIBus Member creation panels.

Figure 23 shows the results for two test runs: one with the transaction logs and file store stored locally on the hard drive, and the other with them stored on a RAMDisk (which essentially treats a section of memory as a hard disk for faster reads and writes). For these runs, a reliability level of express non-persistent was used. The results show that the response times and throughput are slightly faster when the logs are stored on a fast disk.

Figure 23. Performance benefit of using a fast disk

For more information:

Transaction service settings
Summary
IBM WebSphere Application Server has been designed to host an ever-growing range of applications, each with their own unique set of features, requirements, and services. This flexibility affirms the reality that no two applications will use the application server in exactly the same way, and no single set of tuning parameters will likely provide the best performance for any two different applications.

Even though DayTrader might not resemble your application, the methodology for figuring out what to tune and how to go about tuning it is the same. The key is to identify and focus in on the major server components and resources used by your application, based on the application architecture. In general, a large number of applications will realize some improvement from tuning in three core areas: the JVM, thread pools, and connection pools. Other tunables might yield just as much bang for the buck; however, these will typically be tied to a specific WebSphere Application Server feature used by the application.

This article discussed this core set of tunables plus other options that benefit the DayTrader application. For each of the options, you were given information on where to find the tunable, general recommendations or cautions about any trade-offs, and pointers to related tools, if any. Figure 24 shows the overall performance improvement for each DayTrader run time mode after these tuning options were applied:

- JVM heap size
- Thread pool size
- Connection pool size
- Data source statement cache size
- ORB pass by reference
- Servlet caching
- Unlimited persistent HTTP connections
- Large page support
- Disabling PMI.

Figure 24. Overall performance improvements after tuning options applied

As you can clearly see from the figure, the described tuning yielded 169% improvement for EJB mode, 207% improvement for Session Direct mode, and 171% improvement for Direct mode. These are fairly sizable improvements and similar improvements can be realized in other applications; however, you must keep in mind that your results will vary based on the factors discussed earlier, and on other factors that are beyond the scope of this article.
Case study: Tuning WebSphere Application Server V7 and V8 for performance. 

Hopefully, the information and tools discussed here have provided you with some key insights that will help make the task of tuning your application server for your specific applications less daunting and even less intimidating.

Resources

Learn

- Garbage collection in WebSphere Application Server V8, Part 1: Generational as the new default policy
- What's new in WebSphere Application Server V8
- WebSphere Application Server Information Centers (all versions)
- WebSphere Application Server Support

Information Center:

- Tuning the IBM virtual machine for Java
- Performance advisor types and purposes
- Using the performance advisor in Tivoli Performance Viewer
- Tuning application servers
- Data access tuning parameters
- Why you want to use the performance advisors
- Object Request Broker tuning guidelines
- Task overview: Using the dynamic cache service to improve performance
- Configuring cacheable objects with the cachespec.xml file
- HTTP transport channel settings
- Tuning AIX systems
- Tuning Linux systems
- Tuning Solaris systems
- Tuning Windows systems
- Enabling PMI data collection
- Relative advantages of a file store and a data store
- Message reliability levels - JMS delivery mode and service integration quality of service
- Transaction service settings
- addSIBusMember command

Java technology, IBM style: Garbage collection policies, Part 1

- Java Support for Large Memory Pages
- IBM developerWorks WebSphere

Get products and technologies

- Apache DayTrader Benchmark Sample
- IBM WebSphere Application Server V8 trial download
- IBM Support Assistant