JBoss by Red Hat

BEST PRACTICES FOR PERFORMANCE TUNING JBOSS ENTERPRISE APPLICATION PLATFORM 4.3

TIPS AND TRICKS FOR OPTIMIZING YOUR APPLICATION'S PERFORMANCE

2 IN SEARCH OF HIGH-PERFORMANCE APPLICATIONS

2 PERFORMANCE TUNING PRINCIPLES

- 2 Why tune for performance?
- 3 Performance priorities
- 3 A word about performance benchmarks

5 JBOSS EAP 4.3 TUNING

- 5 Connection pooling
- 6 Thread pooling
- 8 Object and component pools
- 9 Logging
- 9 Caching
- 10 JBoss EAP performance tuning summary
- 11 Linux-specific tuning: Large memory pages
- 11 Linux-specific tuning: Tuning the virtual memory manager
- 11 Database tuning update
- 12 Case study

13 APPLICATION PERFORMANCE TUNING: A CONTINUOUS PROCESS

13 APPENDIX: USING LARGE-PAGE MEMORY (LINUX-SPECIFIC INSTRUCTIONS)

IN SEARCH OF HIGH-PERFORMANCE APPLICATIONS

Overall, performance tuning is a very important part of creating, maintaining, and deploying a successful business application. Whether you are building custom applications or deploying commercial, off-the-shelf solutions, you will likely need to tune the application, the database, the middleware, or all three. In fact, 75% of performance issues originate with the application itself.

When organizations select application middleware, performance is always one of their most important selection criteria, if not the most important. In many cases our customers tell us they chose JBoss Enterprise Application Platform (EAP) because of its superior performance. They know that many users of JBoss EAP are achieving superior application performance day after day.

To get the most from your company's investment in middleware, developers and architects need to know the specific ways they can achieve superior performance with JBoss EAP. While we would all like to think |that an application could perform well straight out of the box, this is not usually the case. Applications can have widely varying characteristics, and while some applications might perform well with default middleware settings, others will not.

If you are new to JBoss EAP or performance tuning, this paper will introduce you to best practices that can help you avoid common performance pitfalls as you prepare your application for production. If you're an old hand at application performance issues, you know that technology is constantly changing. You may benefit from an update on best practices in JBoss EAP performance tuning.

PERFORMANCE TUNING PRINCIPLES

WHY TUNE FOR PERFORMANCE?

Performance was once considered just another feature of an application. Today it is frequently considered the most important characteristic of the application—one that can have a significant impact on your business and your productivity. Consider your reaction to a slow web site. If you're like most people, you become frustrated, lose patience, and go elsewhere. If that company is counting on revenue from web sales, it has not only lost your attention—it has lost business. Even for internal applications, poor performance can affect productivity if users have to wait or deal with unpredictable behavior. At best, they might feel annoyed, lose a little time, or form a negative opinion of their IT departments. At worst, business transactions may be lost, or customers may go without important service if users must work around a poorly performing application.

But user experience is not the only reason to tune for performance. A well-performing application will generally use fewer hardware and software resources. A company can optimize its investment in hardware when applications are tuned appropriately, whether that means using older systems longer, purchasing new systems that are more modestly sized, or using fewer systems overall. On the software side, a well-performing application will generally need to use fewer CPU counts or software licenses, no matter what type of software is involved. Reducing software costs can save the company significant money over time.

PERFORMANCE PRIORITIES

Superior performance comes from many layers of the application stack, not just the application server. In many cases, the way the application is designed and how it connects to the database and other software components can have a large impact on overall application performance. Many organizations spend more time tuning their custom-built applications and databases than their underlying application servers. So keep in mind that a superiorly performing application server may have only a minor impact on the overall performance of your applications.

A WORD ABOUT PERFORMANCE BENCHMARKS

Some organizations rely on industry-standard performance benchmarks when selecting middleware. While benchmarks can help vendors, they can deceive you as an evaluator because benchmark applications are usually very different from the applications you will run in production. The current benchmark for application servers, for example, SPECjAppServer2004, is a 2004 application that doesn't take advantage of many recent developments in Java. But aside from that, every application is different and the systems on which benchmark applications are run may be very different from yours. Understanding how a given benchmark runs may tell you very little about how your application will run on your hardware with your settings. Before you rely on a benchmark, you need to investigate the software and hardware configurations used, the server and network configurations, the settings used, the architecture of the application, and how all of these compare with your own environment. We recommend that before selecting a system, you test it with an application and hardware configuration as close to yours as possible.[1]

Performance tuning principle #1:
Understand your performance requirements

The first step in tuning your application for performance is to understand the conditions under which it will need to perform. If your application is a replacement for an existing solution, then your organization already has significant experience with that solution. Chances are you have metrics available such as the number of users, the number of transactions per day, the variations in transaction load or type over the course of a day, week, month, or year, and so on.

If you are deploying a completely new solution, you will need to study that application's business context very carefully. The more you understand exactly how your application will be used, the more successful your performance tuning will be. Sometimes assumptions do not reflect reality. In one case, a development team created and tested its application based on an assumed workload of 60,000 transactions per day. The first day in production, 6 million transactions occurred.

Performance tuning principle #2:
Plan for peaks, not averages

As you examine performance requirements, one of your goals should be to develop a profile of your application's workload with special attention to the peaks. For example, many business applications experience daily peaks in the morning and afternoon with a valley during the middle of the day when

1 For a more in-depth explanation of the pros and cons of performance benchmarks, see Do Performance Benchmarks & Comparisons Matter? – A Guide to Assessing Application Server Performance Results.

employees are eating lunch. Some applications experience peaks at the end of the month, quarter, or season. Your application's workload profile will depend on the specifics of your business, but you should always pay particular attention to periods of peak workload. One of the biggest mistakes developers make is to rely on an average (average daily workload, for example). Averages are not sufficient to ensure that your application will perform during periods of peak load.

Performance tuning principle #3:
Always instrument your application

All applications should be instrumented to provide information for performance analysis. Business conditions, including customer behavior and workload curves, can change dramatically over time, so even if an application once performed well, it may not perform well under today's conditions. If you run into trouble, and your application has not been instrumented for performance, then you have no easy way to know where your problems are. But if your application is appropriately instrumented, then you'll be able to monitor changes in business conditions easily and tune your applications to match before problems occur.

In the past, in order to instrument an application, a developer needed to embed code within the application itself. Today many solutions provide information without requiring developers to code. With JBoss EAP, the call statistics in the container provide you the number of calls, concurrent calls, min call time, max call time, average call time, and the standard deviation of call times. Hibernate statistics provide query execution times, and JBoss ON Monitoring can display all of this information and graph the results in real time. Additionally, numerous third-party application performance management (APM) products are available from certified Red Hat partners to provide application performance information as well.

Finally, for performance-critical situations where one of the other tools doesn't provide the information you need, you can write your own instrumentation using the JBoss AOP framework. This framework offers quite a few features that enable you to see exactly what is happening at runtime. Whether you choose to write your own instrumentation will depend on the application, your skills and available time, and the importance of the application's performance characteristics to its overall success.

Performance tuning principle #4:
Understand where your application spends its time

One important reason to instrument your application is to understand where time is being spent. While you want to know about time across each layer of your application stack, the tools within JBoss EAP will help you understand only part of this equation. If your application is spending too much time in the database, for example, then you may need to focus on your database statistics to pin down the problem.

By understanding where your application spends its time, you will be able to avoid the "shotgun method" of performance tuning—trying multiple solutions to common problems without knowing whether any of them are relevant to your problem. You might solve your problem this way, but the probability is low. Many developers who assume they know what is causing their performance problem are mistaken, resulting in problems that persist for months or sometimes years. That is why having an objective way to know where your application is spending time will help you avoid spending too much time solving performance problems.

Performance tuning principle #5
Replicate or model your production environment

As you work to understand exactly how your application will perform in production, you will need to run it in a test environment. Some companies make it standard practice to implement a test environment that is an exact replica of their production environment. This is an ideal approach in many respects, because developers don't have to wonder how their applications will scale for larger systems.

For many organizations, though, the cost of replicating the production environment is a significant obstacle. Their test environments typically employ smaller systems. If this is your situation, then you will need to create a model that defines the relationship between performance in your test environment and performance in the production environment. That is, you need to model how the application will scale. As you do so, keep the following in mind:

- **Do NOT assume a linear relationship.** Performance doesn't often scale in a linearly. Vendors may tell you they experience linear results, but for real production applications, this is rarely the case.

- **Be conservative in creating your model, and use actual historical data wherever possible.** If you can base your model on data from past experience with other applications, you will be able to refine it over time and increase your confidence in its accuracy.

JBOSS EAP 4.3 TUNING

While 75% of all performance problems are the result of the application, not the middleware or the operating system, you still need to know how to tune settings in the middleware to improve performance and throughput. Depending on your application, one or more of the following may need some attention:

- Connection pooling
- Thread pooling
- Object and component pools
- Logging
- Caching

The following sections provide an overview of each of these areas.

CONNECTION POOLING

Connection pooling and thread pooling are the most important areas to consider when you want to maximize throughput on modern hardware. In terms of system resources, database connections are expensive to set up and tear down. But in spite of this, some applications create a new connection to the database with every query or transaction and then close that connection immediately. This practice adds a great deal of overhead to transaction processing and can lead to poor performance.

To take advantage of the robust connection pooling in JBoss Enterprise Application Platform, start by adjusting connection pool settings on the data source definitions that you can set up in the deploy directory. Set the minimum pool size to the level you want to tune for, then set the maximum at least 25-30% higher. Don't be concerned about setting the maximum too high, as the pool will shrink automatically you don't need that many connections.

To determine the proper sizing, you can monitor your connection usage. Too small a pool will also throttle the application, as the EAP will queue the request for a default of 30,000 milliseconds (or 30 seconds) before giving up and throwing an exception. If you start seeing a lot of 30-second timeouts, that is a strong clue that you need to look at your connection pooling. You can monitor the connection pool utilization from the EAP JMX console, from JBoss ON, or from database-specific tools.

Here is an example of connection pool settings for a data source:

```
<datasources>
 <local-tx-datasource>
  <jndi-name>MySQLDS</jndi-name>
  <connection-url>jdbc:mysql://<host>:3306/Schema</connection-url>
  <driver-class>com.mysql.jdbc.Driver</driver-class>
  <user-name>someuser</user-name>
  <password>somepassword</password>
  <exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.
MySQLExceptionSorter</exception-sorter-class-name>
  <min-pool-size>75</min-pool-size>
  <max-pool-size>100/max-pool-size>
  ...
  <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml -->
  <metadata>
    <type-mapping>mySQL</type-mapping>
  </metadata>
 </local-tx-datasource>
</datasources>
```

THREAD POOLING

Thread pooling is the next most important area to consider as you tune your application for performance. JBoss EAP has robust thread pooling, but before you can size the thread pools appropriately, you need to know how they are used and which ones might be affecting your application's performance. The characteristics of your specific application will determine which thread pools are used and which ones might become bottlenecks. This can vary significantly from application to application. The table below provides a summary of how each thread pool is used.

THREAD POOL	WHERE IS IT DEFINED?	HOW IS IT USED?
System thread pool	In jboss-service.xml in the conf directory	For JNDI naming – the default setting is fine for most cases
HTTPD thread pool in JBoss Web	In the server.xml file under `<server>/deploy/jboss-web.deployer`	When making HTTP requests directly to EAP
AJP thread pool	In the connector section of server.xml	When making HTTP requests through mod_jk
JCA thread pool (also called the Work Manager thread pool)	`<server>/deploy/jbossjca-service.xml`	In conjunction with JMS, as JBoss Messaging uses JCA inflow as the integration into EAP
JBoss Messaging thread pool (for remote clients)	`<server>deploy/jboss-messaging.sar/remoting-bisocket-service.xml`	Pools the TCP sockets
JBoss Messaging thread pool (in JVM clients)	Not directly configurable	Pool is bounded by the number of MDBs you have defined
EJB 3 (same JVM)	Clients in the same JVM will run on whatever thread pool they are already using	For example, a web request comes in through the AJP connector. When it calls an EJB 3 bean, it will continue executing on the AJP connector thread pool.
EJB (remote clients)	`<server>/ejb3/deployer/META-INF/jboss-service.xml`	

Removing connectors

If you are certain that a particular connector will not be used in your application, we recommend removing it. For example, many applications use either the HTTPd thread pool or the mod_jk thread pool but not both. So you can remove the one you don't need.

Monitoring and tuning thread pools

Monitor your thread pools via the EAP 4.3 JMX Console, which displays not only the number of active threads for each pool but also the queue size. We recommend that you adjust thread pool settings through JBoss Operations Network, as it allows you to define settings that persist. Settings adjusted via the JMX console will not survive a reboot. If you use the JMX console, please remember to go back and edit the file if you want the adjustments to be permanent.

Example: HTTP thread pool

```
<Connector port="8080" address="${jboss.bind.address}"
     maxThreads="250" maxHttpHeaderSize="8192"
     emptySessionPath="true" protocol="HTTP/1.1"
     enableLookups="false" redirectPort="8443" acceptCount="100"
     connectionTimeout="20000" disableUploadTimeout="true" />
```

Example: mod_jk or AJP thread pool

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
  <Connector port="8009" address="${jboss.bind.address}" protocol="AJP/1.3"
     emptySessionPath="true" enableLookups="false" redirectPort="8443"
     maxThreads="200" />
```

Example: JCA thread pool

```
<mbean code="org.jboss.util.threadpool.BasicThreadPool"
     name="jboss.jca:service=WorkManagerThreadPool">
   <!-- The name that appears in thread names -->
   <attribute name="Name">WorkManager</attribute>
   <!-- The maximum amount of work in the queue -->
   <attribute name="MaximumQueueSize">1024</attribute>
   <!-- The maximum number of active threads -->
   <attribute name="MaximumPoolSize">100</attribute>
   <!-- How long to keep threads alive after their last work (default one minute)
-->
   <attribute name="KeepAliveTime">60000</attribute>
 </mbean>
 <mbean code="org.jboss.resource.work.JBossWorkManager"
     name="jboss.jca:service=WorkManager">
   <depends optional-attribute-name="ThreadPoolName">jboss.jca:service=
WorkManagerThreadPool</depends>
   <depends optional-attribute-name="XATerminatorName">jboss:service=
TransactionManager</depends>
 </mbean>
```

OBJECT AND COMPONENT POOLS

Object pools and component pools are essentially the same thing. Their settings represent the number of object instances. For EJB 3 two types of pools are defined in `<server>/deploy/ejb3-interceptors-aop.xml`. These are the ThreadLocalPool and the StrictMaxPool. By default, Stateless Session and Stateful Session Beans use the ThreadLocalPool, which is backed by an InfinitePool with no maximum size. Therefore, it grows according to volume in your application. This has the distinct advantage of not needing to be tuned. By default, Message Driven Beans (MDBs) use the StrictMaxPool. This pool actually obeys a maximum, will

queue up requests when that maximum has been reached, and will time out anything in the queue if there is not an available reference from the pool. In this case, the system will throw an exception and if the problem occurred in mid-transaction, you will experience a transaction rollback. Given the impact failed transactions can have on your business, we recommend that you monitor the StrictMaxPool closely via the JMX console.

LOGGING

Developers should take full advantage of logging in the development and testing phases of the application lifecycle. In production, however, logging can cause bottlenecks. You want to be sure that logging provides you with useful information without hurting application throughput. Consider making the following changes as you promote your application into production:

- **Turn off console logging in production.** In EAP's default configuration, console logging is enabled, which means you have the opportunity to see all the logs from your IDE. In production, this is an expensive process with unbuffered I/O. While some applications may be fine with console logging, high-volume applications benefit from turning it off. JBoss EAP 4.3 offers a new configuration set, named Production, which gives developers a better starting point for creating a production environment. In the Production configuration, console logging is turned off.
- **Turn down logging verbosity.** The less you log, the less I/O will occur, and the better your overall application throughput will be. Logging is always a tradeoff, so think carefully about how much logging you really need in production.
- **Use asynchronous logging.** This can make a big difference for high-throughput applications. With asynchronous logging, log messages will go into a queue and control returns to the application as if the logging had been completed. Then a separate thread executes the log operations from the queue.
- **Wrap debug log statements with `If(debugEnabled())`.** This simple practice can make a huge difference if your application contains a lot of debug log statements. Without this condition set, your application creates all of the string objects for each of the log statements, and Log4j creates the LoggingEvent object for each log statement regardless of the log level that is set because the log level is checked only after all of these objects have been created. In some cases this can lead to creation of thousands and thousands of temporary String and LoggingEvent objects, resulting in memory and garbage collection issues and reducing throughput dramatically. By placing a conditional wrapper around your debug log statements, you can ensure that unnecessary log processing does not affect your throughput.

CACHING

Caching, while often very helpful, may be one of the trickiest areas to tune correctly. JBoss Cache is an integral part of the EAP infrastructure, and your application can use it to cache anything you like. Caching is especially valuable for applications that perform a lot of read operations against data that is either completely static or doesn't change frequently, such as reference data. For applications with heavy use of write operations, caching may simply add overhead without providing any real benefit. If not configured properly, Caching may actually reduce throughput.

One of the easiest potential performance enhancements you can make is to cache EJB 3 entities. To define which entities you want cached, modify the file `persistence.xml` that you deploy with your EJB 3 application (an example is shown below). You define the cache size and eviction policy in the file `ejb3-entity-cache-service.xml` found in the deploy directory. These definitions may require some trial and error to get right. Remember that you are working with limited heap space, and a misguided caching strategy can be worse than none at all.

Eviction policy will depend on the specifics of your application. You can define the cache as transactional or read-write. With very large data sets, note that caching may not provide noticeable performance benefits. In this case, shrinking the caches can improve performance. Also, if your application is very write-heavy, it may not benefit from caching. Testing various caching and non-caching configurations will help you determine this.

Example: Settings for caching in persistence.xml

```
<persistence>
<persistence-unit name="services" transaction-type="JTA">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:/MySQLDS</jta-data-source>
    <properties>
      <property name="hibernate.default_catalog" value="EJB3"/>
            ...
      <property name="hibernate.cache.provider_class"
           value="org.jboss.ejb3.entity.TreeCacheProviderHook"/>
      <property name="hibernate.treecache.mbean.object_name"
           value="jboss.cache:service=EJB3EntityTreeCache"/>
      <property name="hibernate.ejb.classcache.services.entities.Customer"
value="read-only"/>
     <property name="hibernate.ejb.classcache.services.entities.Inventory"
value="transactional"/>
           ...
  </properties>
</persistence-unit>
</persistence>
```

JBOSS EAP PERFORMANCE TUNING SUMMARY

To summarize, keep the following performance recommendations in mind when tuning JBoss EAP 4.3.

- Define data sources in the deploy directory and take advantage of EAP's robust connection pooling.
- Understand which thread pools are actually used by your application, remove pools that are not needed, monitor the number of active threads and the queue size, and adjust pool size if needed.
- If you are using message-driven beans, monitor the StrictMaxPool closely to ensure that the maximum object pool size is not a bottleneck.

- Employ different logging strategies for development and production. Especially for high-volume applications, turn console logging off, reduce logging verbosity, use asynchronous logging, and always wrap debug log statements with `If (debugEnabled ())`.
- Take advantage of caching if your application is read-heavy or makes significant use of reference data. Avoid caching for very large data sets or write-heavy applications.

LINUX-SPECIFIC TUNING: LARGE MEMORY PAGES[2]

For today's 64-bit systems, use of large memory pages can improve performance significantly. The default memory page size is typically 4 KB. When you are addressing large amounts of memory, this quickly adds up to a large number of memory pages—just one gigabyte requires 262,144 memory pages. That's a lot for a system to keep track of, which translates to a lot of system overhead.

Aside from helping you avoid the overhead of mapping so many memory pages, large memory pages on Linux cannot be swapped to disk. This is a major advantage because having your heap space swap to disk can wreck havoc on the performance of your application.

Large page support begins at 2 MB and can run as high as 256 MB on some hardware architectures. These numbers will vary, and you will need to find out the values for your specific server. All the major JVM systems support large memory pages on Linux. Because it can be tricky to set up, we provide some specific instructions in the appendix.

When you use large-page memory, keep in mind that the memory is not available to applications in general. To other applications, your system will look as though it has had memory removed from it, because this memory will be dedicated to your specific application. Refer to the appendix for more specific information about configuring your application to use large memory pages.

LINUX-SPECIFIC TUNING: TUNING THE VIRTUAL MEMORY MANAGER

In Linux, the virtual memory manager is tunable. In some cases, you can achieve performance benefits by changing the settings. You won't be using the file system cache much, and you don't want the system to favor the file system cache over your applications, which can sometimes occur with the default settings.

In `/etc/sysctl.conf` you can set `vm.swappiness` to 1 to prevent applications from being swapped to disk when there is memory pressure.

DATABASE TUNING UPDATE

Modern databases, especially 64-bit, are extremely efficient at caching data. In the early days of 64-bit databases, large buffer sizes would slow performance due to elongated search times, but this is no longer true. In our testing, we have experienced OLTP applications with very large buffer caches that show very good results.

Consider your application's characteristics—especially its read/write ratios—when deciding whether and how to use database caching. The majority of applications we see today are read-intensive, as most applications drive their business logic by reading data from the database. The more read-intensive the application is, the more caching can help. If the application is write-heavy, or the data set is very large (from a data warehouse, for example), then a large cache won't help and may slow down the application.

2 While this section provides Linux-specific settings, the principles of large memory page use are applicable to any 64-bit system.

You should use Direct I/O if your database supports it. Especially with a large cache or a write-intensive workload, you should avoid double buffering with the file system buffer cache. Note that MySQL 5 documentation indicates that queries may slow down by up to a factor of 3 when using DIRECT I/O, but we have not experienced this in cases where a properly sized buffer pool is used. If you are using DIRECT I/O, you should be sure to use the virtual memory setting mentioned earlier. We also suggest using asynchronous I/O if your database supports it.

CASE STUDY

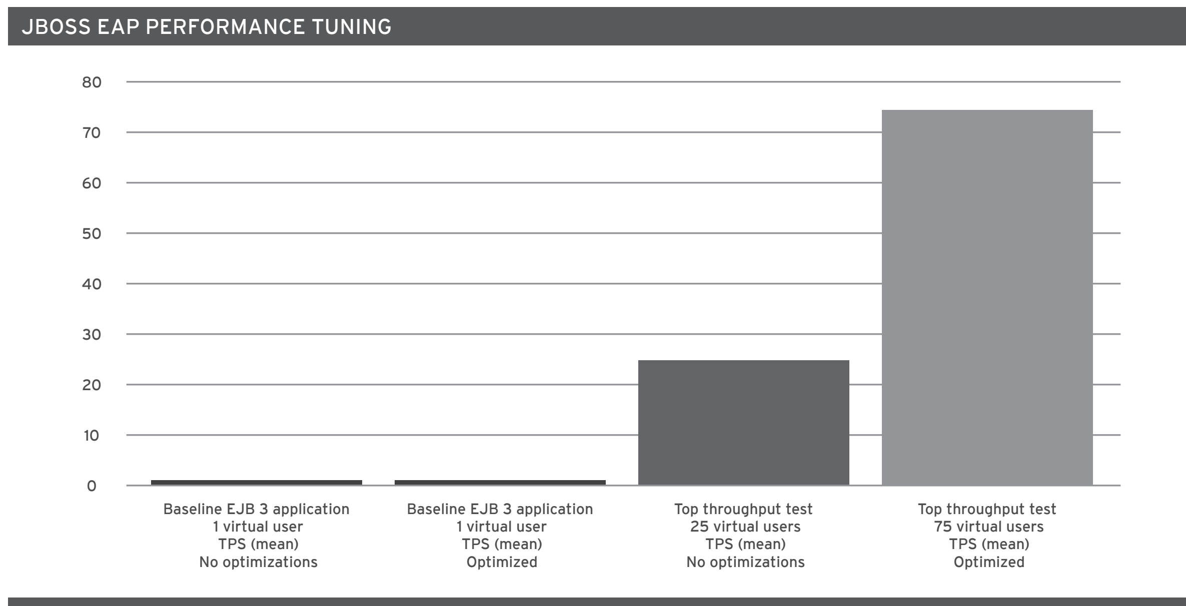
To demonstrate the dramatic results that can be achieved with JBoss EAP performance tuning, we performed an experiment using a sample application.

We began with JBoss EAP's default configuration (with one minor exception), along with all Linux parameters at their defaults. Using Grinder, an open source Java load-testing framework, we measured the highest throughput we could achieve with all of these settings. Using the same application, we then applied many of the optimizations discussed here, both to JBoss EAP and to the operating system (in this case, Linux), and measured throughput again.

The application was an EJB3 application with two servlets for the UI, stateless and stateful session beans for most of the business logic, a message driven POJO for some asynchronous processing, and entities for the persistence. All tests were performed with 3.5 GB heap and a data source with sufficient connections in the pool.

The following graph first illustrates a single virtual user as a baseline for measuring scalability as we added users. As one would expect, the first two bars are statistically equal regardless of the performance optimizations made. The next two bars show how many virtual users we could achieve before we were no longer at 85% of linear scalability, as measured by a transactions per second (TPS) mean from the baseline (the bars on the left). You can see that in the un-optimized case, we were able to reach 25 virtual users, and in the optimized case, we were able to reach 75 virtual users. We achieved a threefold increase in the number of virtual users that the same system could support, simply by optimizing system performance.

JBOSS EAP PERFORMANCE TUNING



APPLICATION PERFORMANCE TUNING: A CONTINUOUS PROCESS

Tuning an application for optimal performance can ensure a positive user experience, promote business productivity, and help to optimize use of hardware and software resources. Performance tuning is not a one-time task, but an ongoing process that ensures a well-performing application as business conditions and system technology change over time. Application developers and architects should always be prepared to tune their applications for performance both before and after they are put into production. As always, the more business-critical an application is and the higher the volume of transactions it must support, the more important performance tuning will be to your business.

When tuned in accordance with the characteristics of your application, JBoss Enterprise Application Platform (EAP) can provide superior application performance. This paper has given you an overview of basic performance principles as well as an introduction to performance tuning best practices for JBoss EAP. Keep in mind that these techniques apply to any of the Red Hat platforms that JBoss EAP supports: JBoss Enterprise Portal Platform, JBoss Enterprise SOA Platform, and JBoss Enterprise BRMS.

For additional information on JBoss EAP performance tuning, please refer to:

- Managing application performance: JBoss Operations Network
- JBoss EAP 4.3 Server Configuration Guide
- JBoss EAP Optimization and Tuning Assistance Services
- JBoss EAP 4.3 Product Documentation

APPENDIX: USING LARGE-PAGE MEMORY (LINUX-SPECIFIC INSTRUCTIONS)

This appendix contains a procedure and example designed to help you take advantage of large-page memory for your applications. See the section "Linux-specific tuning: Large memory pages" earlier in this paper for an overview.

1. Tell the JVM to use large-page memory.

Sun JVM and Open JDK require the following option, passed on the command line, to use large pages:

```
-XX:+UseLargePages
```

The Sun instructions leave it at that, but if you do nothing else you will most likely get the following error:
`Failed to reserve shared memory (error-no=12)`

The following sections describe additional steps you should complete.

2. Set kernel parameters.

Set three kernel parameters in `/etc/sysctl.conf` as follows:

- `kernel.shmmax = n`
  Where n is equal to the number of bytes of the maximum shared memory segment allowed on the system. You should set it to at least the size of the largest heap size you want to use for the JVM. Alternatively, you can set it to the total amount of memory in the system, and you will never have to revisit it.
- `vm.nr_hugepages = n`
  Where n is equal to the number of large pages. You will need to look up the large page size in `/proc/meminfo`.
- `vm.huge_tlb_shm_group = gid`
  Where gid is the ID of a shared group ID for the users you want to have access to the large pages. This setting enables you to limit access to the large memory segment.

3. Set Limits.conf parameters.

Next, set these memlock limits in `/etc/security/limits.conf`:

```
<username> soft memlock n
<username> hard memlock n
```

where `<username>` is the runtime user of the JVM, and n is the number of pages from `vm.nr_hugepages` multiplied by the page size in KB from `/proc/meminfo`.

4. Persist your settings.

Enter the command:

```
sysctl -p
```

This ensures that the settings you created in step 3 will survive a reboot.

5. Reboot.

When the OS allocates these pages, it must find contiguous memory for them or the operation will fail. A reboot will prevent this problem.

6. Confirm page allocation.

When large pages are allocated, `/proc/meminfo` will display a non-zero number for `HugePages_Total`. If you do not see a non-zero number, then you are not using the large pages, and something is configured incorrectly.

We have sometimes seen a problem with MYSQL where the SELinux policy was preventing it from accessing the large pages. Check `/var/log/messages` for `avc_denied` messages (`error-no=13 “permission denied”`) in the mysqld.log.

Example: Setting large memory pages

Consider a server with 8 GB of memory. We will allocate 6 GB to be shared by the JBoss EAP JVM and a MySQL database.

The page size is 2 MB, as shown in `/proc/meminfo (Hugepagesize: 2048KB)`

Configuration: `/etc/sysctl.conf`

- Change maximum shared memory segment size to 8 GB.
  `kernel.shmmax = 8589934592`
- Add the gid to the `hugetlb_shm_group` to give access to the users.
  `vm.hugetlb_shm_group = 501`
- Add 6 GB in 2 MB pages to be shared between the JVM and MySQL.
  `vm.nr_hugepages = 3072`

Calculations:

1024*1024*1024*8 = 8589934592

(1024*1024*1024*6)/(1024*1024*2) or 6 GB/2 MB = 3072 pages

Configuration: `/etc/security/limits.conf`

- Add the limits for memlock to allow the JVM and MySQL to access the large-page memory.

```
jboss       soft  memlock     6291456
jboss       hard  memlock     6291456
mysql       soft  memlock     6291456
mysql       hard  memlock     6291456
root        soft  memlock     6291456
root        hard  memlock     6291456
```

Calculations:

3072 large pages * 2048 KB page size - 3072 *2048 = 6291456

Configuration: `/etc/group`

- Add JBoss and MySQL users to the 501 (`hugetlb`) group in `/etc/group` to give users permission to attach to the shared memory segment.

JBOSS SALES AND INQUIRIES NORTH AMERICA

1-888-REDHAT1
www.jboss.com

Copyright © 2009 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, and RHCE are trademarks of Red Hat, Inc., registered in the U.S. and other countries. Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.