

---

# **Application performance under JBoss 3.2.0**

The effects of the JVM and Java bytecode optimizations

---

**Author:** Jon Barnett

**Document version:** 1.3

## Table of contents

<b>1</b>	<b>Effects of optimizing the JBoss 3.2.0 runtime environment.....</b>	<b>1-1</b>
1.1	Optimizing the JBoss container.....	1-1
1.2	Additional optimizations.....	1-1
1.3	Test setup.....	1-1
1.3.1	JBoss 3.2.0 configuration.....	1-1
1.3.2	Jetty 4.2.9 configuration.....	1-1
1.3.3	Postgresql connection pool.....	1-1
1.3.4	Test application.....	1-1
1.3.5	Test client.....	1-2
1.3.6	Test environment.....	1-2
1.4	Results.....	1-3
1.4.1	Default JBoss 3.2.0 with Linux Sun 1.4.1 JDK.....	1-3
1.4.2	Default JBoss 3.2.0 with Linux IBM 1.4.0 SDK.....	1-3
1.4.3	Optimized JBoss 3.2.0 with Linux Sun 1.4.1 JDK.....	1-3
1.4.4	Optimized JBoss 3.2.0 with Linux IBM 1.4.0 SDK.....	1-3
1.5	Comparison.....	1-3
1.5.1	Medium run (4800 samples).....	1-3
1.5.2	Long run (9600 samples).....	1-4
1.5.3	Analysis.....	1-4
<b>A</b>	<b>Optimizing the JBoss 3.2.0 system.....</b>	<b>A-1</b>
A.1	Building JBoss 3.2.0.....	A-1
A.1.1	Prerequisites.....	A-1
A.1.2	Modifying the build properties.....	A-1
A.1.3	Building the distribution.....	A-2
A.2	Building extra Jetty 4.2.9 components.....	A-2
A.2.1	.....	A-2
A.2.2	Modifying the Jetty build.....	A-2
A.2.3	Building the distribution.....	A-3
A.2.4	Incorporating the classes.....	A-3
A.3	Additional optimizations.....	A-3
A.3.1	Libraries for manual optimizations.....	A-3
A.3.2	Optimizing the libraries.....	A-4
A.3.3	Completing the process.....	A-4
<b>B</b>	<b>Additional observations.....</b>	<b>B-1</b>
B.1	Starting and stopping JBoss 3.2.0.....	B-1
B.1.1	Results.....	B-1
B.1.2	Comparison.....	B-1
B.2	Windows 2000 reference timing.....	B-1
B.2.1	Configuration.....	B-2
B.2.2	Default JBoss 3.2.0 with Windows Sun 1.4.1 JDK.....	B-2
B.2.3	Optimized JBoss 3.2.0 with Windows Sun 1.4.1 JDK.....	B-2

# 1 Effects of optimizing the JBoss 3.2.0 runtime environment

This study is intended to examine the effects of the Java runtime environment and Java bytecode optimizations of the JBoss 3.2.0 code on a J2EE application. It is meant to be indicative of the trends rather than being a comprehensive analysis of the performance gains. It is still a worthwhile exercise to examine the performance of your own applications and identify any possible bottlenecks within your own architecture that would nullify any performance gains provided by these optimizations.

## 1.1 Optimizing the JBoss container

The default JBoss 3.2.0 binaries as distributed from the JBoss website has no bytecode optimizations. Optimizing your bytecode, at least in theory, should improve the speed at which your code executes by removing extraneous instructions and improving the organization of bytecode sequences. The optimizations should also reduce the memory footprint of classes.

. This only performs an optimization of namespace and removes line information and debug information from the classes. However, this is the safest route as some optimization tools remove code that is relevant such as “if then” traps with no “then” code which are used to prevent execution in some long “if then else” chains. This optimization at least reduces the memory footprint and improves calls without risking damage to the operation.

## 1.2 Additional optimizations

Since most J2EE-based applications are intended to access and assemble data, the other obvious point where optimization may be of benefit is with respect to datasource access.

## 1.3 Test setup

### 1.3.1 JBoss 3.2.0 configuration

The JBoss 3.2.0 configuration used in the tests is the default runtime instance. This is largely unchanged except for the removal of the JMX console web application and the UUID key generator. All environments contain the Amity framework components and the Amity suite of products, although none of these were used during the testing.

### 1.3.2 Jetty 4.2.9 configuration

The standard embedded Jetty service for JBoss 3.2.0 was used. However, we compiled the IBM JSSE listener and the socket channel listener to make use of the more scalable standard listener and a secure listener for the IBM JVM. The secure socket listeners were configured but not used in the tests.

### 1.3.3 Postgresql connection pool

The connection pool was configured so that no upper limit was created for the pool that would throttle the application load in any way. However, an optimized Postgresql JDBC driver was used in the optimized JBoss 3.2.0 container whereas the default Postgresql JDBC driver was used with the default JBoss 3.2.0 container.

### 1.3.4 Test application

The application that makes use of the JBoss infrastructure consists of a single page of content. The content is created and assembled through four servlets. These servlets access four data tables in Postgresql to obtain the content.

The application byte code has been optimized, as has the application architecture. These optimizations have remained in place for all tests, to ensure that the application itself would operate as fast as possible.

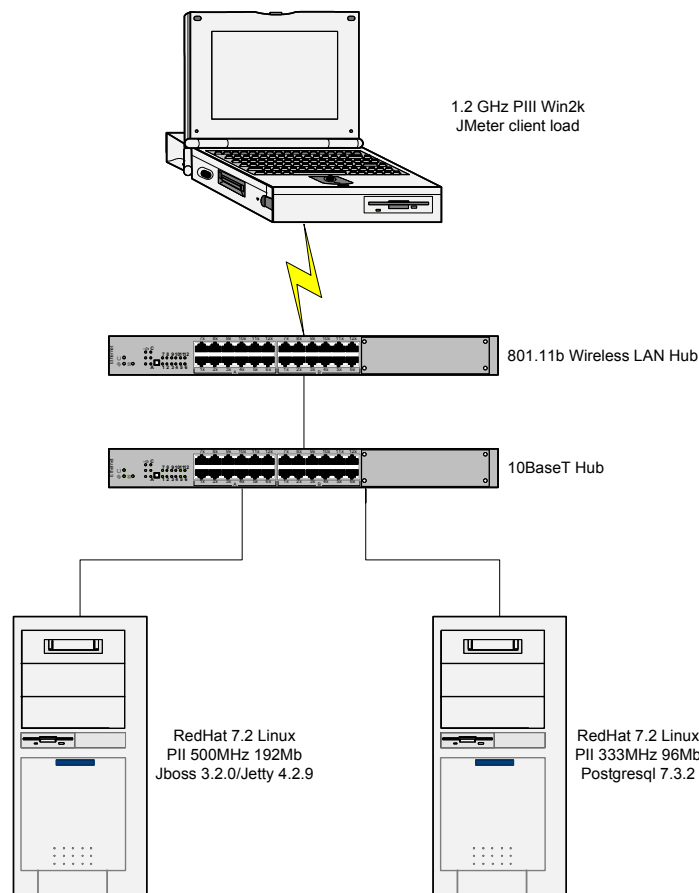
### 1.3.5 Test client

We use Apache JMeter to generate the test load. While we want the load to have sufficient weight, we do not want to create so many requests that the system under test cannot service the load. For our purposes, we decided that 3 requests per second with 48 clients would provide a reasonable test load. This was generated by defining 48 threads, using a uniform random timer with a maximum delay of 30 seconds. Only the servlet content was retrieved, excluding any graphic images, style sheets or other linked content. A large sample space was taken to ensure that statistics generated were not unduly biased by network fluctuations or other events.

We accessed the application using the socket channel listener so that any differences between the IBM JSSE listener and the Sun JSSE listener had no bearing on the results.

### 1.3.6 Test environment

The test environment features a fairly modest server environment with some fixed delays. Improvements in the network speeds and the database server would reduce these fixed delays to the response time and most likely would amplify the relative magnitude of the performance increases measured. However, the measurements are intended to give an indication of the possible performance improvements rather than being an absolute measure.



## 1.4 Results

### 1.4.1 Default JBoss 3.2.0 with Linux Sun 1.4.1 JDK

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
4800	175 ms	80 ms	7240 ms	2.8 / sec	7	8
9600	197 ms	70 ms	64973 ms	2.7 / sec	7	13

VSZ	RSS
292400 kb	155944 kb

### 1.4.2 Default JBoss 3.2.0 with Linux IBM 1.4.0 SDK

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
4800	179 ms	60 ms	16694 ms	2.8 / sec	9	19
9600	155 ms	60 ms	16694 ms	2.7 / sec	9	10

VSZ	RSS
202476 kb	133120 kb

### 1.4.3 Optimized JBoss 3.2.0 with Linux Sun 1.4.1 JDK

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
4800	196 ms	80 ms	14611 ms	2.9 / sec	13	13
9600	195 ms	80 ms	65064 ms	2.7 / sec	12	13

VSZ	RSS
312252 kb	123500 kb

### 1.4.4 Optimized JBoss 3.2.0 with Linux IBM 1.4.0 SDK

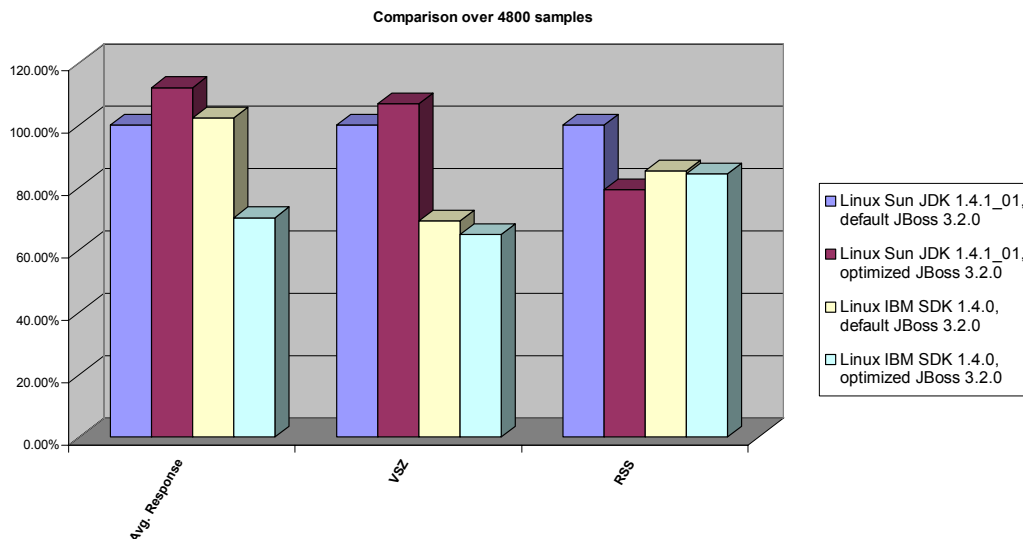
Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
4800	123 ms	50 ms	2554 ms	2.8 / sec	7	7
9600	125 ms	50 ms	2734 ms	2.8 / sec	7	8

VSZ	RSS
189824 kb	131624 kb

## 1.5 Comparison

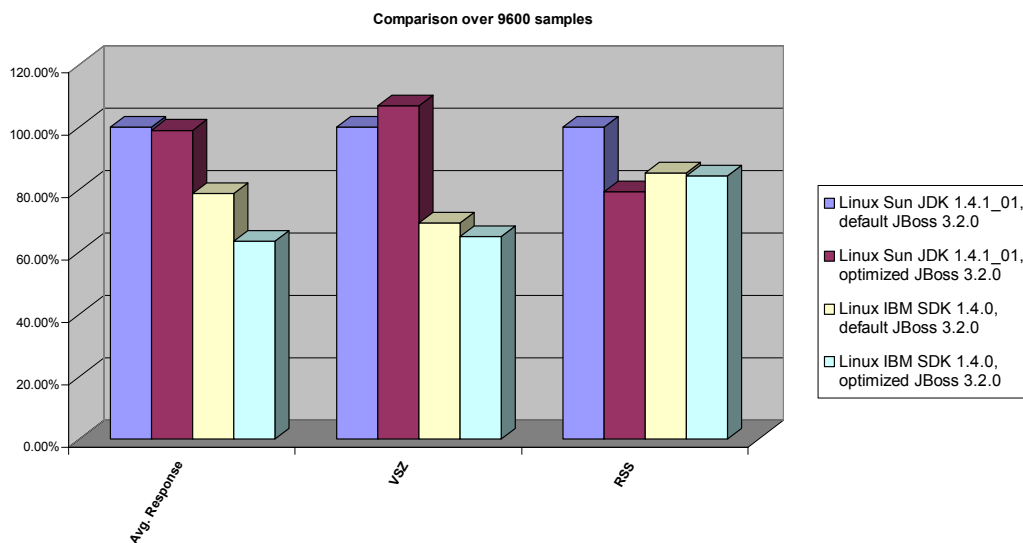
### 1.5.1 Medium run (4800 samples)

System	Avg. Response	VSZ	RSS
Linux Sun JDK 1.4.1_01, default JBoss 3.2.0	175 ms	292400 kb	155944 kb
Linux Sun JDK 1.4.1_01, optimized JBoss 3.2.0	112.00 %	106.79 %	79.20 %
Linux IBM SDK 1.4.0, default JBoss 3.2.0	102.29 %	69.25 %	85.36 %
Linux IBM SDK 1.4.0, optimized JBoss 3.2.0	70.29 %	64.92 %	84.40 %



### 1.5.2 Long run (9600 samples)

System	Avg. Response	VSZ	RSS
Linux Sun JDK 1.4.1_01, default JBoss 3.2.0	197 ms	292400 kb	155944 kb
Linux Sun JDK 1.4.1_01, optimized JBoss 3.2.0	98.98 %	106.79 %	79.20 %
Linux IBM SDK 1.4.0, default JBoss 3.2.0	78.68 %	69.25 %	85.36 %
Linux IBM SDK 1.4.0, optimized JBoss 3.2.0	63.45 %	64.92 %	84.40 %



### 1.5.3 Analysis

From the perspective of performance, optimizing bytecode does not seem to have much effect when using the Sun JDK. In some cases the application response suffered under the optimized code. However under the IBM SDK, optimizing the bytecode improved the performance by reducing the response time between 31.28 and 19.36 percent in our test case.

The IBM SDK produced a response time reduction of between 21.32 and -2.29 percent over the Sun JDK using the default bytecode version of JBoss 3.2.0.

The bytecode optimization of JBoss 3.2.0 combined with the IBM Java 1.4.0 runtime environment reduces the response time between 36.55 and 29.71 percent over the combination of the default bytecode version of JBoss 3.2.0 and the Sun Java 1.4.1 runtime environment.

Optimization seems to adversely affect the virtual size of the JBoss system when using the Sun JDK. With the IBM SDK, the virtual size is reduced between 35.08 and 30.75 percent over the default bytecode version of JBoss 3.2.0 in a Sun JDK environment.

In all cases, the portion of the JBoss system in memory is improved through a combination of optimization and the use of the IBM SDK. However with the IBM SDK, the effect is less pronounced as the JVM seems to determine the optimal code segments to remain in memory independent of the bytecode optimization. The IBM JVM produces better results with optimized bytecode than the Sun JVM.

## A Optimizing the JBoss 3.2.0 system

Most of the JBoss environment can be easily optimized as the Ant build scripts for the distribution provide the capability. However, some class libraries in JBoss are from other projects and are not touched by the JBoss build process. For these libraries, you will need to provide your own optimizations.

Additionally, we have found that because of the choice for JBoss 3.2.0 to support Java 2 version 1.3.x upwards, the distribution does not include some scalability components such as a non-blocking socket listener in the Jetty service. It also doesn't support the IBM JSSE listener required if you use the IBM SDK. Since we are interested in production ready builds, we also included these in our preparation work.

The complexity of JBoss 3.2.0 means that the distribution suffers from breakages if you run a post-optimizer. This is the reason we use the native optimization in the JBoss 3.2.0 build process and only perform minimal post-optimizations.

### A.1 Building JBoss 3.2.0

For the most part, you only need to modify the primary build script for JBoss 3.2.0 and use your choice of compiler to create an optimized distribution.

#### A.1.1 Prerequisites

You will need Ant to run the build script. You will also need a compiler capable of optimizing the bytecode. Unfortunately, at this point in time, Jikes does not support bytecode optimizations. For our tests, we used Sun JDK 1.4.1\_01 on Windows 2000.

#### A.1.2 Modifying the build properties

Unpack the JBoss 3.2.0 source distribution. The section in *build/local.properties* of your distribution will control the Java compiler for our needs. For the default build, the settings should be:

```
### General compiler configuration ###

#build.compiler=jikes
#build.warnings=true
#build.pedantic=true
#javac.depend=on

### Javac/Jikes compiler configuration ###

javac.optimize=off
javac.debug=on
javac.deprecation=on
```

For an optimized build, the settings should be:

```
### General compiler configuration ###

#build.compiler=jikes
#build.warnings=true
#build.pedantic=true
#javac.depend=on

### Javac/Jikes compiler configuration ###

javac.optimize=on
javac.debug=off
javac.deprecation=on
```

### A.1.3 Building the distribution

From the build directory of the source distribution, start a build compilation after ensuring that the existing build has been cleaned. In Windows, you can run the build/build.bat script.

```
build clean
build
```

In Linux/Unix, you would run the build/build.sh shell script.

```
./build.sh clean
./build.sh
```

The binary distribution will be created in the build/output directory of the source distribution.

## A.2 Building extra Jetty 4.2.9 components

In order to build the extra Jetty components such as the socket channel listener and the IBM JSSE listener, you will need to download the Jetty 4.2.9 source from SourceForge at [http://sourceforge.net/project/showfiles.php?group\\_id=7322](http://sourceforge.net/project/showfiles.php?group_id=7322).

### A.2.1

You will need a Java 2 JDK 1.4.0 or higher to compile the socket channel listener as it requires the non-blocking IO routines introduced by the 1.4.x release. You will need the IBM JSSE library from an IBM JRE/JDK 1.3.x release or higher in order to build the IBM JSSE listener. Place the *ibmjsse.jar* in the *ext* library of the Jetty 4.2.9 source distribution.

### A.2.2 Modifying the Jetty build

In order to build the IBM JSSE library, you will need to modify build.xml in the top most directory of the Jetty 4.2.9 source distribution. The contrib. section should be modified to the following:

```
<target name="contrib" depends="prepare,classes"
  description="Compile the contrib classes" >

  <!-- <available property="IBM.available"
    classname="com.ibm.jsse.IBMJSSEProvider"
    classpathref="extpath"/>-->
  <property name="IBM.available" value="true"/>
  <available property="tyrex.available"
    classname="tyrex.naming.MemoryContext"
    classpathref="extpath" />
```

I used this to avoid the need for any messy fiddling to detect the IBM JSSE classes.

Modify the build script to perform the optimizations you require. A default build with no optimizations would be:

```
<property name="build.compiler"           value="modern" />
<property name="build.compiler.emacs"     value="true" />
<property name="build.compiler.fulldepend" value="false" />
<property name="build.compiler.pedantic"  value="false" />
<property name="javac.debug"              value="on" />
<property name="javac.optimize"           value="off" />
<property name="javac.deprecation"        value="off" />
```

An optimized build would be modified to the following:

```

<property name="build.compiler"           value="modern" />
<property name="build.compiler.emacs"     value="true" />
<property name="build.compiler.fulldepend" value="false" />
<property name="build.compiler.pedantic"  value="false" />
<property name="javac.debug"              value="off" />
<property name="javac.optimize"           value="on" />
<property name="javac.deprecation"        value="off" />

```

### A.2.3 Building the distribution

In order to build the JDK 1.4.x non-blocking IO socket channel listener, you only need to use the JDK 1.4 compiler. The build script will detect the environment and build the additional classes. Build the binaries including the IBM JSSE listener by performing the following ant build commands:

```

ant clean
ant
ant contrib

```

The socket channel listener classes are located in the directory, *classes1.4* of the source distribution. The IBM JSSE listener is in *classes/org/mortbay/http/IbmJsseListener.class*.

### A.2.4 Incorporating the classes

The easiest way of moving these into the standard JBoss libraries is to take the *org.mortbay.jetty.jar* file from the JBoss 3.2.0 build and adding the additional classes found under the *classes1.4* directory and the *org/mortbay/http/IbmJsseListener.class* file from the Jetty 4.2.9 build, ensuring the directory structure of all these classes is preserved. Make sure the modified *org.mortbay.jetty.jar* replaces all instances in the JBoss build – existing in *deploy/jbossweb-jetty.sar* of the various JBoss instances.

## A.3 Additional optimizations

For pre-built classes used with the JBoss distribution, we have performed some code optimizations with Retroguard. We believe it provides the best and most configurable optimization processing. jopt is the oldest optimizer but the optimizations result in larger libraries than obtained using jarg and Retroguard. jarg creates the smallest libraries but sometimes creates code execution problems with even simple classes – constant static variables are sometimes mangled. In our experience, Retroguard falls between the other two optimizers in terms of code size and does not suffer from any code execution problems.

We supply an ant build script for helping with the optimization and the command files that instruct Retroguard on how to perform the optimizations.

### A.3.1 Libraries for manual optimizations

The following is a list of Java libraries that benefit from manual optimization:

- activation.jar
- ant.jar
- axis.jar
- axis-ant.jar (only used if you are have an Axis install, not a JBoss.NET install)
- commons-discovery.jar
- commons-httpclient.jar
- commons-logging.jar
- jasper-compiler.jar
- jasper-runtime.jar
- jaxrpc.jar
- jts.jar
- log4j.jar
- log4j-boot.jar

- mail.jar
- pg73jdbc3.jar (only if you are using Postgresql as your datasource)
- saaj.jar
- webdavlib.jar
- xalan.jar
- xercesImpl.jar

### A.3.2 Optimizing the libraries

We have built an ant script that helps optimize these additional libraries we have used in the testing. The optimizations are constructed so that all public and protected declarations are preserved. Remember that these are namespace optimizations and no bytecode should be harmed by the process.

Take the distribution we provide and unpack it. Download the Retroguard obfuscator from [Retrologic](#) and place retroguard.jar in the *ext* directory of the distribution. There should be some other libraries already in there from the JBoss 3.2.0 build. If you are using another JBoss 3.2.x distribution we suggest that you should update this directory with the files from your distribution. We can't guarantee the optimizer will work with other JBoss 3.2.x distributions, but the scripts will give you a starting point for your own optimizations.

The original files for namespace optimization and debug information removal are placed in the *org* directory of the distribution. For the optimizations to operate correctly, we suggest you place all the mandatory files that we list in the *org* directory.

The configuration files for the Retroguard process are contained in the *conf* directory of the distribution and are of the form *module.rgs* where *module* is the name of the file to be cleaned.

Completed libraries are put in the *opt* directory.

Problems with the build of a particular module are noted in the log directory in *module.out.log* and *module.log*.

Run the optimization from the root directory of the distribution using the following:

```
ant -Dmodule=library-name build
```

where *library-name* is the name of the library that you wish to optimize.

For example, to optimize the log4j.jar,

```
ant -Dmodule=log4j build
```

This will search for a log4j.jar in the *org* directory and optimize it using the Retroguard instructions log4j.rgs in the *conf* directory.

### A.3.3 Completing the process

Once you have finished optimizing all these libraries, you can check these against the unoptimized libraries. They should be smaller than the originals. Also check the namespace optimization logs to ensure that no problems have occurred. Some warnings may appear but these are usually references to external libraries, indicating that these methods should not be obfuscated.

The libraries we have listed benefit from this footprint optimization. We have checked the remainder of the libraries included with JBoss 3.2.0 and have found that they have already been optimized and suggest that you may harm these files by running this process on them.

Once you are satisfied that the optimizations have completed successfully, replace existing instances in the JBoss distribution with these libraries. You should now have an optimized JBoss 3.2.0 distribution.

## B Additional observations

There are a few observations related to the testing conducted that do not have any impact on the analysis. They are however worth noting for those interested in adopting any of the measures used here.

### B.1 Starting and stopping JBoss 3.2.0

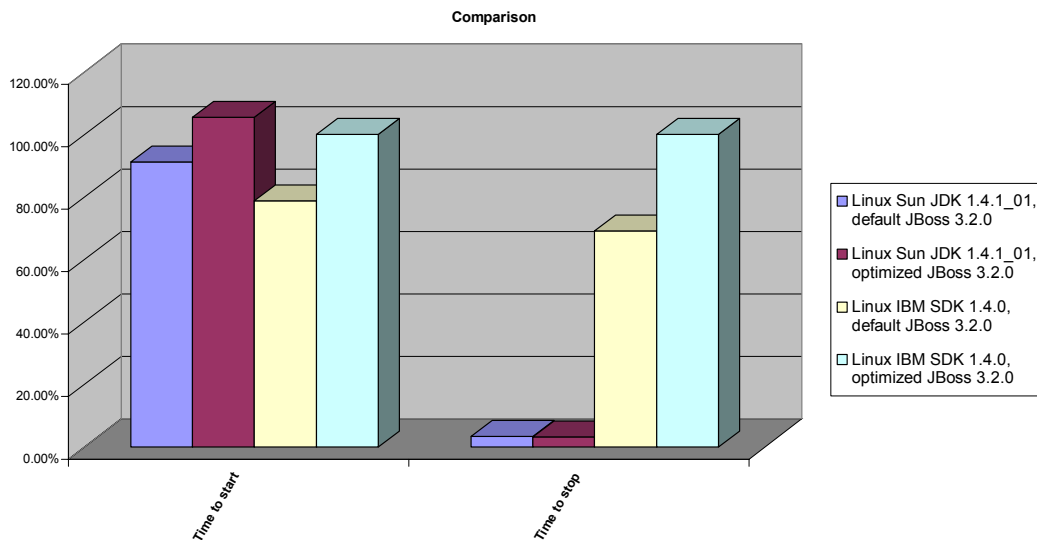
The IBM SDK 1.4.0 is faster than the Sun JDK 1.4.1\_01 when starting JBoss 3.2.0 and is much slower when shutting. Optimizing JBoss bytecode slows the start up and in the case of the IBM SDK, slows the shut down process.

#### B.1.1 Results

System	Time to start	Time to stop
Linux Sun JDK 1.4.1_01, default JBoss 3.2.0	4m 54s 793ms	0m 07s 561ms
Linux Sun JDK 1.4.1_01, optimized JBoss 3.2.0	5m 41s 059ms	0m 07s 100ms
Linux IBM SDK 1.4.0, default JBoss 3.2.0	4m 14s 383ms	2m 34s 627ms
Linux IBM SDK 1.4.0, optimized JBoss 3.2.0	5m 23s 086ms	3m 43s 499ms

#### B.1.2 Comparison

The comparison uses the optimized JBoss 3.2.0 service with the IBM SDK 1.4.0 as the benchmark.



### B.2 Windows 2000 reference timing

In order to provide some reference timings for our own benefit, we performed some tests where all the components are located on the same machine using a different operating system to Linux. We had hoped to be able to run the tests against the IBM SDK 1.4.0 but at the time of writing, this version has not been released for the Windows platform. However, we provide the measurements we have taken for reference.

### B.2.1 Configuration

A Microsoft Windows 2000 PIII 1.2GHz system with 384 Mb of memory was used in these tests. The same JBoss 3.2.0 configurations as used in the Linux tests were employed. Postgresql 7.3.2 running on the same platform within a Cygwin shell was used as the database. JMeter with the same programmed test load was operated from the same Windows environment. The overall configuration removed network delays in the test system. The speed of the system and the relatively low load was expected to ensure that the test results were not overly biased by task swapping.

### B.2.2 Default JBoss 3.2.0 with Windows Sun 1.4.1 JDK

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
4800	40 ms	10 ms	841 ms	2.9 / sec	7	7
9600	42 ms	10 ms	851 ms	2.8 / sec	7	7

### B.2.3 Optimized JBoss 3.2.0 with Windows Sun 1.4.1 JDK

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
4800	44 ms	10 ms	2013 ms	2.8 / sec	7	9
9600	42 ms	10 ms	2013 ms	2.8 / sec	7	9