BEA WebLogic JRockit™

Migration Guide

Version 8.1 Service Pack 2
December 2003

Contents

Introduction

Welcome to the BEA WebLogic JRockit *Migration Guide*, a guide for Java developers creating applications destined to run with BEA WebLogic JRockit JVM and then migrate them from third-party JVMs to BEA WebLogic JRockit. This document contains this information:

- The best coding practices developers should follow to ensure optimal performance with any JVM, particularly WebLogic JRockit. See Best Coding Practices for JVM Migration.
- Adetailed troubleshooting guide that will lead you through the solutions to some of the more common problems developers have encountered with WebLogic JRockit. See Troubleshooting.
- A description of some popular debugging and profiling tools available with WebLogic JRockit SDK. See Profiling and Debugging with WebLogic JRockit.
- Procedures for migrating applications developed on a third-party JVM to WebLogic JRockit. See Migrating to WebLogic JRockit.

Best Coding Practices for JVM Migration

This section contains guidelines for writing applications to run on WebLogic JRockit SDK. This information provided here is in no way complete; it merely helps you avoid some common pitfalls. BEA Systems does not want to compromise Java's "write once run everywhere" notion. On the contrary, this sections highlights guidelines that are valid for any Java program. They are, however, especially important when switching between JVMs in general and between Sun Microsystem's HotSpot JVM and WebLogic JRockit JVM in particular.

The best coding practices are summarized in the following subjects:

- Read the Relevant Specifications
- Never Use Deprecated Unsafe Methods
- Minimize the Use of Finalizers
- Don't Depend on Thread Priorities
- Don't Use Internal sun.* or COM.jrockit.* Classes
- Override java.Object.hashCode for User Defined Classes When Using java.util.Hashtable
- Do Careful Thread Synchronization
- Expect Only Standard System Properties
- Minimize the Number of Java Processes
- Avoid Calling System.gc()

Read the Relevant Specifications

Read the Java language specification and Java API specification carefully and do not rely on unspecified behavior.

A WebLogic JRockit JVM is based on a number of specifications; for example, The Java Virtual Machine Specification and the Java API Specification. You should be aware that many implementations of these specifications exist: WebLogic JRockit is one. You should never expect any particular behavior that is *not* specified in one of these documents. Unspecified behavior might differ between the Sun JVM and WebLogic JRockit JVM. Note too that behavior is sometimes different between individual releases of the Sun JVM and can also change between releases of WebLogic JRockit JVM.

You can find these specifications at the following sites:

- The Java Language Specification

  `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`
- Java 2 Platform API Specification,

  `http://java.sun.com/j2se/1.4.1/docs/api/index.html`
- The Java Virtual Machine Specification

  `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

The specifications are written to give JVM vendors freedom to optimize their JVMs, and therefore they leave certain behavior unspecified. You should understand, however, that numerous parts of the specifications mentioned above are unspecified. The following examples describe three of these unspecified elements.

Example 1: Reflection

The Java API Specification of the method `getMethods()` on the `java.lang.Class` class clearly states: “The elements in the array returned are not sorted and are not in any particular order.”

Example 2: Reflection Revisited

The `toString()` method of the `java.lang.reflect.Method` might include the access modifier `native`. Therefore, you should not rely on the result of this call to be equal between JVM implementations. Some classes in the Java API specification are implemented as `native` either by WebLogic JRockit JVM and the Sun JVM. There is no guarantee that a native implementation on one JVM has to be native on another one.

Example 3: Serialization

The Java API Specification of the method `defaultReadObject()` of the `java.lang.ObjectInputStream` class does not specify the order in which fields are de-serialized; hence no such order can be expected.

Never Use Deprecated Unsafe Methods

The deprecated methods

- `java.lang.Thread.stop`
- `java.lang.Thread.suspend`
- `java.lang.Thread.resume`
- `java.lang.Runtime.runFinalizersOnExit`

are inherently unsafe, and should never be used. For more information see:

`http://java.sun.com/j2se/1.4/docs/guide/misc/threadPrimitiveDeprecation.html`

Minimize the Use of Finalizers

Finalizers are often error prone since they often implicitly depend on the order of execution. This order differs amongst JVMs, and between consecutive runs on the same JVM. Using finalizers is also inherently bad for performance since it imposes an additional burden on the memory management system that needs to handle execution of finalizers and let objects live longer. For more information on using—and not using—finalizers, please refer to:

http://access1.sun.com/techarticles/weak.references.html

http://www.memorymanagement.org/glossary/r.html#reference.object

Don't Depend on Thread Priorities

Be careful when using `java.lang.Thread.setPriority`. Depending on thread priorities might lead on unwanted or unexpected results since the scheduling algorithm might choose to starve lower priority threads of CPU time and never execute them. Furthermore the result might differ between operating systems and JVMs.

The Java API specification states that "Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority."

The priority set by the `setPriority()` method is a parameter that might be used in the thread-scheduling algorithm, which shares CPU execution time between executing threads. This algorithm might be controlled either by the JVM or by the operating system. It is important to be aware of the fact that this algorithm normally differs between operating systems and that the algorithm might change between releases of both the operating system and the JVM. For WebLogic JRockit JVM native threads, the algorithm is implemented by the operating system; for thin threads it is implemented by the WebLogic JRockit JVM (it is partly controlled by the operating system since thin threads runs on top of operating system threads or lightweight processes).

Don't Use Internal sun.* or COM.jrockit.* Classes

The classes that WebLogic JRockit SDK includes fall into package groups `java.*`, `javax.*`, `org.*`, `sun.*` and `COM.jrockit.*`. All but the `sun.*` and `COM.jrockit` packages are a standard part of the Java platform and will be supported into the future.In general, non-standard packages, which are outside of the Java platform, can be different across JVM vendors and OS platforms (Windows, Linux, and so on) and can change at any time without notice with SDK versions. Programs that contain direct usage of the `sun.*` and `COM.jrockit.*` packages are not 100% Pure Java.

For more information, please refer to the note about `sun.*` packages at:

`http://java.sun.com/products/jdk/faq/faq-sun-packages.html`

Override java.Object.hashCode for User Defined Classes When Using java.util.Hashtable

On WebLogic JRockit, the current default implementation of hashCode returns a value for the object determined by the JVM. The value is created using the memory address of the object. However, because this value can be reused if the object is moved during garbage collection, it is possible to obtain the same hash code for two different objects. Also, two objects that represent the same value are guaranteed to have the same hash code only if they are the exact same object. This implementation is not particularly useful for hashing; therefore, derived classes should override `hashCode()`.

Do Careful Thread Synchronization

Make sure that you synchronize threads that access shared data. Synchronization bugs often appear when changing JVMs because the implementation of locks, garbage collection, thread scheduling and so on, might differ significantly.

Expect Only Standard System Properties

When implementing `java.lang.System.getProperties()` or `java.lang.System.getProperty()`, you should only depend on standard system properties being returned; different VMs may return a different set of extended properties. Non-standard properties should not be returned.

When the JVM starts, it inserts a number of standard properties into the system properties list. These properties, and the meaning of their values, are listed in the Java API specification. Do not expect any other non-standard properties.

`http://java.sun.com/j2se/1.4/docs/api/java/lang/System.html#getProperties()`

Minimize the Number of Java Processes

When designing applications there is sometimes a choice between running several processes, i.e. JVM instances versus running several threads or thread groups within a single process, i.e. JVM instance. If possible, it is more effective to use as few JVM instances as possible per physical machine.

Avoid Calling System.gc()

Don't call `java.lang.System.gc()`. This method behaves differently with WebLogic JRockit than with other JVMs. Instead of doing a complete garbage collection, as with the Sun JVM and others, when called by an application running with WebLogic JRockit, `System.gc()` behaves depending upon the garbage collector already in use:

- If you're using a generational copy collector, `System.gc()` does a collection in their nursery.
- For all other collectors, it does a collection only if one is needed. In other words, it does nothing special on the call.

The WebLogic JRockit JVM garbage collector will generally do a much better job of deciding when to do garbage collection than will `System.gc()`. If you are having problems with memory usage, pause times for garbage collection, and so on, you are better off configuring the WebLogic JRockit JVM memory management system appropriately. See uning WebLogic JRockit JVM in the *Tuning WebLogic JRockit 8.1 JVM*.

Troubleshooting

While the process of switching to WebLogic JRockit **8.1** JVM from another JVM is relatively easy and generally problem-free, you might encounter some known issues while or after making this switch. This section describes some of those issues and describes some simple workarounds. The issues that might occur are:

- An Application Does Not Run
- Slow-to-Start Applications
- Large Memory Consumption
- Slow Performance vis-a-vis HotSpot
- Randomly Appearing Bugs
- WebLogic JRockit Throws Errors HotSpot Does Not Throw
- Slow Performance in Development Mode
- WebLogic JRockit Does Not Run Jakarta Tomcat as a Windows Service
- Support for the /3GB Windows Startup Option
- Support for PAE on Windows
- Other Frequently Asked Questions

An Application Does Not Run

I cannot get my favorite Java application to run on WebLogic JRockit. What am I doing wrong?

Many problems with running applications on WebLogic JRockit is due to erroneous environment variables or non-standard startup options.

Start by ensuring that your environment variables are set up correctly. Make sure that you have set your `JAVA_HOME` environment variable correctly, i.e. set to the directory where WebLogic JRockit has been installed, and that `"%JAVA_HOME%\bin"` is available in your `PATH` environment variable before any other directory where any version of java.exe may exist. When running applications as Windows services it is crucial that you set these environment variables system wide. To do this:

1. Open the Start menu and select Settings>Control Panel>System
2. Select the Advanced tab.
3. Click Environment Variables.
4. To set system wide environment variables you must edit in the lower part of this dialog box, labeled System variables.

Applications are often started via scripts. Make sure that none of the startup scripts includes non-standard startup options for java. See *Tuning WebLogic JRockit 8.1 JVM* for complete documentation of standard and non-standard options.

Slow-to-Start Applications

Why does it take longer for my applications to start with WebLogic JRockit?

Java programs are compiled into byte code by a Java compiler. Many JVMs, including the Sun JVM, interprets this byte code each time it is executed. WebLogic JRockit, however, uses code generation technology to generate native machine code from the byte code. This is sometimes called Just-In-Time (JIT) compilation. The code generation step imposes an initial time penalty before execution. Normally, the subsequent execution of the code is faster than interpreting the byte code. WebLogic JRockit is optimized for server applications that normally run for long periods of times. The initial time penalty is normally negligible in comparison to the performance gains of code generation over time.

Large Memory Consumption

Why does my applications consume more memory when running on WebLogic JRockit?

The Java programming language relies on a mechanism called garbage collection (GC) to free memory when it is no longer being used. There is no equivalence to the `delete` operator in the C++ programming language or the `free` function in the C programming language. Any Java virtual machine must include a garbage collector that handles the task of finding unreferenced objects, possibly invoke their finalizers and free the memory used to hold their state.

The WebLogic JRockit garbage collectors are described in Selecting and Running a Memory Management System in *Using WebLogic JRockit 8.1 SDK*. Generally, the WebLogic JRockit garbage collection implementations trade high memory usage for speed and minimal program wide halts; that is, acquiring system wide locks.

Slow Performance vis-a-vis HotSpot

I have a script/program that use WebLogic JRockit for certain tasks. Why is it slower than when I use the Sun JVM, HotSpot?

This might be related to the Slow-to-Start Applications above. Scripts or other programs may start many Java processes and may therefore experience bad performance compared to the Sun JVM, since WebLogic JRockit has a code generation penalty when starting up. When starting many Java processes and running them only for a short time, this penalty can become significant.

Randomly Appearing Bugs

Why does my application have randomly appearing bugs when running on WebLogic JRockit that it doesn't have when running on the Sun JVM?

You may be experiencing synchronization bugs in your application. It is not uncommon that such bugs are revealed when switching JVMs. The Java Virtual machine specification and the Java language specification leaves plenty of room for optimization that may cause unsynchronized access to shared data, to cause different behavior on different JVMs.

For more information see:

`http://java.sun.com/docs/books/jls/second_edition/html/memory.doc.html`

`http://java.sun.com/docs/books/vmspec/2nd-edition/html/Threads.doc.html`

WebLogic JRockit Throws Errors HotSpot Does Not Throw

Why is WebLogic JRockit throwing `IllegalAccessError`, `ClassFormatError`, `IncompatibleClassChangeError` or other `LinkageError` exceptions when the Sun JVM is not?

Verification ensures that the binary representation of a class or interface is structurally correct. For example, it checks that every instruction has a valid operation code; that every branch instruction branches to the start of some other instruction, rather than into the middle of an instruction; that every method is provided with a structurally correct signature; and that every instruction obeys the type discipline of the Java programming language.

If an error occurs during verification, then an instance of the following subclass of class `LinkageError` will be thrown at the point in the program that caused the class to be verified

Example: Using JTidy throws `IllegalAccessError`

In an early version of JTidy from Apache Software Foundation, the compiler had incorrectly inlined a reference to a private variable belonging to an outer class. This caused an exception to be thrown since WebLogic JRockit does stricter verification than the Sun JVM is. The old `Tidy.jar` should be replaced with the new and correctly compiled version.

Slow Performance in Development Mode

Why is WebLogic JRockit slower when WebLogic Server is running in development mode?

When WebLogic Server is started in development mode, WebLogic JRockit is by default started with the `-Xdebug` option. This makes WebLogic JRockit run with some overhead.

Note: This option is purely for diagnostics use and should therefore not be used in a production type environment.

WebLogic JRockit Does Not Run Jakarta Tomcat as a Windows Service

I cannot get WebLogic JRockit to run Jakarta Tomcat as a Windows service. What am I doing wrong?

The quick answer to this problem is: if you are using `jk_nt_service`, do everything that you need to do for the Sun JVM, then exchange the non-standard Sun JVM `-Xrs` startup option with the non-standard WebLogic JRockit `-Xnohup` in the `wrapper.properties` configuration file. The rest of this answer is a slightly more detailed description of this.

1. First make sure that you have completed all the tasks concerning the environment variables in “I cannot get my favorite Java application to run on WebLogic JRockit. What am I doing wrong?” above.

   Many people use the `jk_nt_service` windows service wrapper to run java applications; for example, Jakarta Tomcat, as a Windows service (see http://members.ozemail.com.au/~lampante/howto/tomcat/iisnt/). Independently of what Windows service you may be using you must make sure that it is not using any non-standard startup options. When using `jk_nt_service`, the startup is defined in:

```
<tomcat install dir>\conf\jk\wrapper.properties
```

2. Make sure that you set the three properties `wrapper.tomcat_home`, `wrapper.java_home` and `wrapper.cmd_line` are set accordingly:
   - `wrapper.tomcat_home` must be set with the installation directory of tomcat
   - `wrapper.java_home` must be set to the same value as the `JAVA_HOME` environment variable.
   - The property `wrapper.cmd_line` defines the startup command. At the time of writing, this property should be set to:

     ```
     wrapper.cmd_line=$(wrapper.javabin) -Xnohup
     -Djava.security.policy=="$(wrapper.tomcat_policy)"
     -Dtomcat.home="$(wrapper.tomcat_home)" -classpath
     $(wrapper.class_path) $(wrapper.startup_
     ```

     for WebLogic JRockit. Normally this command includes a non-standard option to stop the JVM from shutting down the process when a user logs off. For WebLogic JRockit this non-standard option is `-Xnohup`, for the Sun JVM it is `-Xrs`.

Support for the /3GB Windows Startup Option

Does WebLogic JRockit support the /3GB Windows startup option?

This version of WebLogic JRockit does not support the /3GB option. It will be supported in the next release. I you want to use this feature—with the understanding that it **is not** supported, use the Microsoft Visual Studio tool, `editbin.exe`, on `java.exe` to enable it.

Support for PAE on Windows

Does WebLogic JRockit support Physical Address Extension (PAE) on Windows?

Physical Address Extension provides the possibility to map physical memory into your process virtual memory address space. At any given time, you can still have just 2 GB mapped (3 GB with the /3GB option; see Support for the /3GB Windows Startup Option for limitations of this option with this version of WebLogic JRockit); that is, you have to re-use a portion of the virtual address space to map different portions of the physical address space. Therefore you cannot simply address more memory with regular pointers. This does not work well with a normal Java heap, which uses 32-bit pointers to reference objects and needs the entire heap in the virtual address space at all times.

PAE works well with applications that can jointly control mapping and memory access. At this point this is not well suited for Java. However a native database driver might use PAE to cache data. You should discuss with the various database vendors how to make this happen (if it is not available already). You might even create a custom cache by rolling your own “DB access layer” on top of a native database API using JNI.

Other ways of using your plentiful physical memory includes using several Java processes (each Windows process can get 2 GB physical memory to use for its private address space).

If you a need a larger Java heap, you need to use a 64-bit implementation of WebLogic JRockit; that is, the IA-64 versions for Windows 2003 or Linux.

Other Frequently Asked Questions

What is a Java virtual machine?

As described by the Java Virtual Machine Specification:

“The Java virtual machine is the cornerstone of the Java and Java 2 platforms. It is the component of the technology responsible for its hardware- and operating system- independence, the small size of its compiled code, and its ability to protect users from malicious programs.”

What is the difference between the Sun JVM and WebLogic JRockit?

The most well know JVM is the implementation from Sun. The Sun JVM is called HotSpot. The Sun JVM is shipped in the Java Developer's Kit (JDK) and Java Runtime Environment (JRE) from Sun.

The WebLogic JRockit JVM from BEA systems is optimized for reliability and performance for server side applications. To achieve this, WebLogic JRockit uses technologies such as code generation, hot spot detection, code optimization, advanced garbage collection algorithms and tight operating system integration.

Should I write my applications differently for WebLogic JRockit?

No! You should not write your applications in any other way for WebLogic JRockit than you should for any other JVM. You should, however, design and implement your applications well in order for them to run well on WebLogic JRockit.

Should I use thin threads with BEA WebLogic Server?

We don’t recommend it. Thin threads is an alternative to the traditional 1:1 thread mapping model. It may offer improved scalability and performance if used with an application with many (for example, greater than 100 threads); you may want to try this threading implementation if your application is so characterized. Please note, however, that thin threads is experimental in this release.

CHAPTER 4

Profiling and Debugging with WebLogic JRockit

WebLogic JRockit SDK includes the JVM profiling interface (JVMPI) and JVM debugging interface (JVMDI), interfaces that enable Java applications to interact with the JVM to assist with profiling and debugging activities. While developers will need to implement these interfaces within their application code, users' exposure to JVMPI and JVMDI will usually be through the profiling and debgging tools they select for the applications they are running.

This section includes information on the following subjects:

- Profiling WebLogic JRockit
- Debugging with WebLogic JRockit

Profiling WebLogic JRockit

You can use any number of third-party profiling tools to profile WebLogic JRockit performance. This section describes how to use the Java Virtual Machine Profiler Interface (JVMPI) to facilitate using those tools.

Using JVMPI

The JVM Profiler Interface allows you to use third-party profiling tools with WebLogic JRockit SDK.

Warning: This interface is an experimental feature in the Java 2 SDK and is not yet a standard profiling interface.

How JVMPI Works

JVMPI is a two-way function call interface between the Java virtual machine and an in-process profiler agent. On one hand, the VM notifies the profiler agent of various events, corresponding to, for example, heap allocation, thread start, and so on. Concurrently, the profiler agent issues controls and requests for more information through the JVMPI. For example, the profiler agent can turn on/off a specific event notification, based on the needs of the profiler front-end.

The profiler front-end may or may not run in the same process as the profiler agent. It may reside in a different process on the same machine, or on a remote machine connected via the network. The JVMPI does not specify a standard wire protocol. Tools vendors may design wire protocols suitable for the needs of different profiler front-ends.

A profiling tool based on JVMPI can obtain a variety of information such as heavy memory allocation sites, CPU usage hot-spots, unnecessary object retention, and monitor contention, for a comprehensive performance analysis.

JVMPI supports partial profiling; that is, a user can selectively profile an application for certain subsets of the time the VM is up and can also choose to obtain only certain types of profiling information.

Limitations

JVMPI has the following limitations:

- The current version of JVMPI supports only one agent per VM.
- You can profile with JVMPI only if you use a generational copy garbage collector (`-Xgc:gencopy`), *unless* you set the `-Xjvmpi:allocs=off` flag. In that case, you can use any garbage collector.

Changing the JVMPI Default Behavior

Use the following option to modify the JVMPI default behavior:

```
-Xjvmpi [:<argument1>=<value1>[,<argumentN>=<valueN>]]
```

When WebLogic JRockit runs with a profiling agent attached, by default a number of events are enabled that can create significant overhead. Since JVMPI doesn’t require all of these events to be sent, you can disable them by setting the `-Xjvmpi` flag. Use the arguments listed in Table 4-1 to modify the default behavior.:

Table 4-1 Command Line Arguments for `-Xjvmpi`

Argument	Description
`entryexit=off`|`on` (default `on`)	Setting this to `off` disables the following method entry and exit events sent by JVMPI: • `JVMPI_EVENT_METHOD_ENTRY` • `JVMPI_EVENT_METHOD_ENTRY2` • `JVMPI_EVENT_METHOD_EXIT`
`allocs=off`|`on` (default `on`)	Setting this to `off` disables these object allocation and free events: • `JVMPI_EVENT_OBJECT_ALLOC` • `JVMPI_EVENT_OBJECT_MOVE` • `JVMPI_EVENT_OBJECT_FREE` • `JVMPI_EVENT_ARENA_NEW` • `JVMPI_EVENT_ARENA_DELETE`
`monitors=off`|`on` (default `on`)	Setting this to `off` disables these monitor contention events: • `JVMPI_EVENT_RAW_MONITOR_CONTENDED_ENTER` • `JVMPI_EVENT_RAW_MONITOR_CONTENDED_ENTERED` • `JVMPI_EVENT_RAW_MONITOR_CONTENDED_EXIT` • `JVMPI_EVENT_MONITOR_CONTENDED_ENTER` • `JVMPI_EVENT_MONITOR_CONTENDED_ENTERED` • `JVMPI_EVENT_MONITOR_CONTENDED_EXIT` • `JVMPI_EVENT_MONITOR_WAIT` • `JVMPI_EVENT_MONITOR_WAITED`
`arenadelete=off`|`on` (default `off`)	Setting this to `on` will enable the `JVMPI_EVENT_ARENA_DELETE` event. This event is suppressed by default to be compatible with Sun's VM which does not send this event. The event can be enabled if a profiler wishes to receive the event.

Additional JVMPI Documentation

As JVMPI is an experimental interface, Sun Microsystems provides the documentation for tools vendors who have an immediate need for profiling hooks in the Java VM. You can find this documentation at:

http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/

Debugging with WebLogic JRockit

This section describes the interface by which debugging tools can interface with WebLogic JRockit to debug Java applications.

Java Virtual Machine Debugger Interface (JVMDI)

JVMDI is a low-level debugging interface used by debuggers and other programming tools. It allows you to inspect the state and to control the execution of applications running in the WebLogic JRockit JVM.

JVMDI describes the functionality a JVM provides to enable debugging of Java applications running within the JVM. JVMDI defines the services a JVM must provide for debugging. JVMDI services include requests for information (for example, current stack frame), actions (set a breakpoint), and notification (when a breakpoint has been hit).

How JVMDI Works

JVMDI is a two-way interface:

- The JVMDI client can be notified of interesting occurrences through events.
- The JVMDI can query and control the application through many different functions, either in response to events or independent of them.

JVMDI clients run in the same VM as the application being debugged and access JVMDI through a native interface. The native, in-process interface allows maximum control with minimal intrusion of a debugging tool. Typically, JVMDI clients are relatively compact. They can be controlled by a separate process that implements the bulk of a debugger's functionality without interfering with the target application's normal execution.

JVMDI Documentation

Sun Microsystems provides complete reference documentation for the Java Platform Debug Architecture and JVMDI. For more information, go to:

http://java.sun.com/j2se/1.4.1/docs/guide/jpda/index.html

CHAPTER 5

Migrating to WebLogic JRockit

This section describes how to migrate Java applications developed on another JVM to WebLogic JRockit so that they perform to their optimal capability when running on BEA WebLogic Server. It contains information on the following subjects:

- About Application Migration
- Migration Procedures
- Testing the Application
- Submitting Migration Tips

About Application Migration

Migrating an application to BEA WebLogic JRockit JVM is a relatively simple process, requiring some minor environmental changes and following some simple coding guidelines. This section provides instructions and tips to successfully completing this simple process. It also describes some of the benefits and possible problems you might encounter during migration and it discusses some best J2SE coding practices for you to follow to ensure that your application runs successfully once it is running on WebLogic JRockit.

Why Migrate?

BEA WebLogic JRockit JVM is the default JVM shipped with BEA WebLogic Server. Although there are other JVMs available on the market today that you can use to develop Java applications, BEA Systems recommends that you use WebLogic JRockit JVM as the production JVM for any application deployed on WebLogic Server.

Migration Restrictions

Migration is available only for Intel-based Windows systems and Linux systems. For a list of supported platforms, please refer to:

`http://edocs.bea.com/wljrockit/docs81/certif.html`

Migration Support

Should you experience any problems or find any bugs while attempting to migrate an application to WebLogic JRockit 8.1, please send us an e-mail at **support@bea.com**. We would appreciate if you could provide as much information as possible about the problem, for example:

- Hardware
- Operating system and its version
- The program you are attempting to migrate
- Stack dumps (if any)
- A small code example that will reproduce the error
- Copies of any `*.dump` and `*.mdmp` files (`*.mdmp` files are available only on Windows)

Migration Procedures

This section describes basic environmental and implementation changes necessary to migrate to WebLogic JRockit JVM from Sun Microsystems HotSpot JVM or any other third-party JVM. It includes information on the following subjects:

- Environment Changes
- Other Tips
- Tuning WebLogic JRockit JVM for Your Application
- Testing the Application
- Submitting Migration Tips

Environment Changes

To migrate from HotSpot (or any third-party JVM) to WebLogic JRockit JVM, you need to make the following changes to the files.

- Set the `JAVA_HOME` environmental variable in `<WEBLOGIC_HOME>/common/commEnv.cmd` (or .sh) to the appropriate path.
- Set the `JAVA_VENDOR` environmental variable in `<WEBLOGIC_HOME>/common/commEnv.cmd` (or .sh) to `BEA`.
- If you are using a start-up script, remove any Sun-specific (or other JVM provider) options from the start command line (like `-hotspot`). If possible, replace them with WebLogic JRockit-specific options; for example, `-jrockit`. Other flags that might need to be changed include MEM_ARGS and JAVA_VM.
- Change `config.xml` to point the default compiler setting(s) to the WebLogic JRockit `javac` compiler.

Other Tips

For information on other coding practices that will ensure a successful migration of your application to WebLogic JRockit JVM, please refer to Best Coding Practices for JVM Migration.

Tuning WebLogic JRockit JVM for Your Application

Once you've migrated your application to WebLogic JRockit JVM, you might want to tune the JVM for optimal performance. For example, you might want to specify a different start-up heap size or set custom garbage collection parameters. For more information on tuning WebLogic JRockit JVM, please refer to the *Tuning WebLogic JRockit 8.1 JVM*.

The non-standard options, that is, options preceded with `-X`, are critical tools for tuning a JVM at startup. These options change the behavior of WebLogic JRockit JVM to better suit the needs of different Java applications.

While all JVMs use non-standard options, the option names might not be the same from JVM to JVM; for example, while WebLogic JRockit JVM will accept the non-standard option `-Xns` to set the nursery in generational concurrent and generational copying garbage collectors, Sun's HotSpot JVM uses the option `-XX:NewSize` to set this value.

If you are migrating an application to WebLogic JRockit, we recommend that you become familiar with the non-standard options available to you. For more information, please refer to Appendix A: Command Line Options by Name, in *Tuning WebLogic JRockit 8.1 JVM*.

You should also be aware that, being non-standard, non-standard options are subject to change at any time.

Testing the Application

Always test your application on WebLogic JRockit JVM before putting it into production. If you develop your application on the Sun JVM (HotSpot), you *must* test your application on JVM before you put it into production.

Why Test?

Some important reasons for testing are:

- Sometimes you might find bugs in your own program that don't occur on the Sun JVM; for example, synchronization problems.
- You might have used third party class libraries that are not 100% Java and rely on Sun-specific classes or behavior.
- You might have used third-party class files that are not correct. WebLogic JRockit has been known to enforce verification more rigorously than the Sun JVM.

How to Test

To test your application on WebLogic JRockit:

1. Run your application against any test scripts or benchmarks that are appropriate for that application.
2. If any problems occur, handle them as you normally would for the specific application.

Submitting Migration Tips

The migration tips discussed in this section represent an evolving list. Often, a successful migration to WebLogic JRockit depends as much upon the application being migrated as it does to the VMs being used. BEA Systems welcomes suggestions based upon your experiences with migrating applications to WebLogic JRockit. Feel free to submit any migration ideas or comments to the WebLogic JRockit SDK migration newsgroup at:

```
jrockit.developer.interest.migration
```