IBM WebSphere Real Time for AIX
Version 3

User Guide

IBM WebSphere Real Time for AIX
Version 3

User Guide

Note
Before using this information and the product it supports, read the information in "Notices" on page 67.

Fifth edition (February 2014)

This edition of the user guide applies to IBM WebSphere Real Time for AIX, Version 3, and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 2003, 2014.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2003, 2014

Figures

© Copyright IBM Corp. 2003, 2014

Tables

© Copyright IBM Corp. 2003, 2014

Preface

This user guide provides general information about IBM® WebSphere® Real Time for AIX®.

© Copyright IBM Corp. 2003, 2014

Chapter 1. Introduction

This information tells you about IBM WebSphere Real Time for AIX.

Any new modifications made to this user guide are indicated by vertical bars to the left of the changes.

Late breaking information about the IBM WebSphere Real Time for AIX that is not available in the user guide can be found here: http://www.ibm.com/support/docview.wss?uid=swg21501145

Overview of WebSphere Real Time for AIX

WebSphere Real Time for AIX bundles real-time capabilities with the IBM J9 virtual machine (JVM).

WebSphere Real Time for AIX is a Java™ Runtime Environment with a Software Development Kit that extends the IBM SDK for Java with real-time capabilities. Applications that are dependent on precise response times can take advantage of the real-time features provided with WebSphere Real Time for AIX on standard Java technology.

Features

Real-time applications need consistent run time rather than absolute speed.

The main concerns when deploying real-time applications with traditional JVMs are as follows:
- Unpredictable (potentially long) delays from Garbage Collection (GC) activity.
- Delays to method run time as Just-In-Time (JIT) compilation and recompilation occurs, with variability in execution time.
- Arbitrary operating system scheduling.

WebSphere Real Time for AIX removes these obstacles by providing:
- The Metronome Garbage Collector, an incremental, deterministic garbage collector with very short pause times.

What's new

This topic introduces changes for IBM WebSphere Real Time for AIX .

WebSphere Real Time for AIX V3

WebSphere Real Time for AIX V3 is an extension to the IBM SDK for Java V7, building on the features and functions available with this release to include real-time capabilities. Earlier versions of WebSphere Real Time for AIX were based on earlier releases of the IBM SDK for Java.

© Copyright IBM Corp. 2003, 2014

To learn more about what's new in IBM SDK for Java V7, see: What's new in the IBM SDK for Java 7 information center.

Any new modifications made to this user guide are indicated by vertical bars to the left of the changes.

Controlling pause times for the Metronome Garbage Collector

By default, the metronome garbage collector pauses for 3 milliseconds between garbage collection cycles. You can change this value to control the pause time using a new command-line option. For more information about this option, see "Controlling pause time" on page 17.

Compressed references

The metronome garbage collector now supports uncompressed references as well as compressed references on 64-bit platforms. For any performance implications, see Chapter 7, "Performance," on page 25.

Benefits

The benefits of the real-time environment are that Java applications run with a greater degree of predictability than with the standard JVM and provide consistent timing behavior for your Java application. Background activities, such as compilation and garbage collection, occur at given times and thus remove any unexpected peaks of background activity when running your application.

You obtain these advantages by extending the JVM with the Metronome real time garbage collection technology.

Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

IBM strives to provide products with usable access for everyone, regardless of age or ability.

For example, you can operate WebSphere Real Time for AIX without a mouse, by using only the keyboard.

To read about issues that affect accessibility of the underlying IBM SDK for Java V7, see IBM Information Center. There are no accessibility issues affecting unique features and capabilities in WebSphere Real Time for AIX.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

For users who require keyboard navigation, a description of useful keystrokes for Swing applications can be found here: Swing Key Bindings.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility.

Chapter 2. Understanding IBM WebSphere Real Time for AIX

This section introduces key components of IBM WebSphere Real Time for AIX.

- "Introduction to the Metronome Garbage Collector"

Introduction to the Metronome Garbage Collector

The Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for AIX.

The key difference between Metronome garbage collection and standard garbage collection is that Metronome garbage collection occurs in small interruptible steps but standard garbage collection stops the application while it marks and collects garbage.

For example:

```
java -Xgcpolicy:metronome -Xgc:targetUtilization=80 yourApplication
```

The example specifies that your application runs for 80% in every 60ms. The remaining 20% of the time might be used for garbage collection, if there is garbage to be collected. The Metronome Garbage Collector guarantees utilization levels provided that it has been given sufficient resources. Garbage collection begins when the amount of free space in the heap falls below a dynamically determined threshold.

Metronome garbage collection and class unloading

Metronome supports class unloading in the same way as a standard Java developer kit. However, because of the work involved, while unloading classes there might be pause time outliers during garbage collection activities.

Metronome Garbage Collector threads

The Metronome Garbage Collector consists of two types of threads: a single alarm thread, and a number of collection (GC) threads. By default, GC uses one thread for each logical active processor available to the operating system. This enables the most efficient parallel processing during GC cycles. A GC cycle means the time between GC being triggered and the completion of freeing garbage. Depending on the Java heap size, the elapsed time for a complete GC cycle could be several seconds. A GC cycle usually contains hundreds of GC quanta. These quanta are the very short pauses to application code, typically lasting 3 milliseconds. Use **-verbose:gc** to get summary reports of cycles and quanta. For more information, see: "Using verbose:gc information" on page 54. You can set the number of GC threads for the JVM using the **-Xgcthreads** option.

There is no benefit from increasing **-Xgcthreads** above the default. Reducing **-Xgcthreads** can reduce overall CPU load during GC cycles, though GC cycles will be lengthened.

Note: GC quanta duration targets remain constant at 3 milliseconds.

You cannot change the number of alarm threads for the JVM.

© Copyright IBM Corp. 2003, 2014

The Metronome Garbage Collector periodically checks the JVM to see if the heap memory has sufficient free space. When the amount of free space falls below the limit, the Metronome Garbage Collector triggers the JVM to start garbage collection.

Alarm thread

The single alarm thread guarantees to use minimal resources. It "wakes" at regular intervals and makes these checks:

- The amount of free space in the heap memory
- Whether garbage collection is currently taking place

If insufficient free space is available and no garbage collection is taking place, the alarm thread triggers the collection threads to start garbage collection. The alarm thread does nothing until the next scheduled time for it to check the JVM.

Collection threads

The collection threads perform the garbage collection.

After the garbage collection cycle has completed, the Metronome Garbage Collector checks the amount of free heap space. If there is still insufficient free heap space, another garbage collection cycle is started using the same trigger ID. If there is sufficient free heap space, the trigger ends and the garbage collection threads are stopped. The alarm thread continues to monitor the free heap space and will trigger another garbage collection cycle when it is required.

For more information about using the Metronome Garbage Collector, see "Using the Metronome Garbage Collector" on page 17.

Chapter 3. Planning

Read this section before installing WebSphere Real Time for AIX.

•
• "Supported environments"
•
• "Considerations" on page 7

Migration

You can run your standard Java applications on WebSphere Real Time for AIX without modification.

Supported environments

IBM WebSphere Real Time for AIX is supported on certain hardware platforms and operating systems.

IBM WebSphere Real Time for AIX

The 32-bit and 64-bit SDKs run on hardware that supports the following platform architectures:

- IBM POWER® 4
- IBM POWER 5
- IBM POWER 6
- IBM POWER 7
- JS20 blades

IBM WebSphere Real Time for AIX also runs on older System p® systems that have a Common Hardware Reference Platform (CHRP) architecture. To test whether IBM WebSphere Real Time for AIX is supported on a specific System p system, at the system prompt type:

```
lscfg -p | fgrep Architecture
```

The output for a supported platform reads:

```
Model Architecture: chrp
```

A minimum of 512 MB of physical memory is required for simple applications. For good performance, more complex applications require a larger memory configuration.

IBM WebSphere Real Time for AIX operates on Very Large Symmetric Multiprocessor systems. However, the additional computational power of systems with more than eight physical processor cores might give diminishing benefits. To optimize the extra capacity of these systems, multiple LPARs with up to eight physical processors each are recommended.

The following operating systems are supported:

© Copyright IBM Corp. 2003, 2014

Table 1. AIX environments tested

Operating system	32-bit SDK	64-bit SDK
AIX 6.1 TL5	Yes	Yes
AIX 7.1.0.0	Yes	Yes

Additional information for AIX

Important information for IBM WebSphere Real Time for AIX.

AIX APARs required for IBM WebSphere Real Time for AIX.

To avoid problems when using Java, ensure that you have any prerequisite AIX APARs installed. For further information about the APARs needed for an AIX level, see http://www.ibm.com/support/docview.wss?uid=swg21605167.

Environment variables

The environment variable **LDR_CNTRL=MAXDATA** is not supported for 64-bit processes. Use **LDR_CNTRL=MAXDATA** only on 32-bit processes.

Graphics terminal

If you are using IBM WebSphere Real Time for AIX on 64-bit AIX, with UTF-8 locale and the local graphics terminal uses the UTF-8 locale, you might see an exception from `java.io.Console`.

On AIX 6.1, the exception is:

```
IZ97736: CANNOT CONTROL TTY ATTRIBUTE BY USING 64BIT PROGRAM
```

For more information, see the APAR https://www-304.ibm.com/support/docview.wss?uid=isg1IZ97736.

On AIX 7.1, the exception is:

```
IZ97912: CANNOT CONTROL TTY ATTRIBUTE BY USING 64BIT PROGRAM
```

For more information, see the APAR https://www-304.ibm.com/support/docview.wss?uid=isg1IZ97912.

Use of non-UTF8 CJK locales

If you are using one of the supported non-UTF8 CJK locales, you must install one of these file sets.

```
X11.fnt.ucs.ttf (for ja_JP or Ja_JP)
X11.fnt.ucs.ttf_CN (for zh_CN or Zh_CN)
X11.fnt.ucs.ttf_KR (for ko_KR)
X11.fnt.ucs.ttf_TW (for zh_TW or Zh_TW)
```

Note: The installation images are available on the AIX base CDs. Updates are available from the AIX fix distribution website.

When using the `zh_TW.IBM-eucTW` locale on 64-bit AIX 6.1, you might get a result that uses `ISO-8859-1` instead of `IBM-eucTW`, in response to the following command:

```
$ LANG=zh_TW locale charmap
```

The different return might affect operation of the IBM WebSphere Real Time for AIX. If you encounter this effect, contact IBM Support for more information.

Java 2D graphics

If you want to use the improved Java 2D graphics pipeline, based on the X11 XRender extension, you must install the `libXrender.so` library, version 0.9.3 or later.

Considerations

You must be aware of a number of factors when using WebSphere Real Time for AIX.

- Where possible, do not run more than one real-time JVM on the same system. The reason is that you would then have multiple garbage collectors. Each JVM does not know about the memory areas of the other. One effect is that GC cycles and pause times cannot be coordinated across JVMs, meaning that it is possible for one JVM to affect adversely the GC performance of another JVM. If you must use multiple JVMs, ensure that each JVM is bound to a specific subset of processors by using the **execrset** command.
- The shared caches used by earlier WebSphere Real Time for AIX releases to store precompiled code and classes are not compatible with the caches used by this release of WebSphere Real Time for AIX. You must regenerate the contents of the earlier caches.
- When using shared class caches, the cache name must not exceed 53 characters.
- Workload partitions (WPAR) are not supported.
- Micropartitions are not supported. Logical partitions (LPAR) with an integral number of processors are supported, but LPARs with a fractional number of processors, for example 0.5 or 1.5, are not.

Chapter 4. Installing IBM WebSphere Real Time for AIX

Follow these steps to install WebSphere Real Time for AIX.

Installation files

You require these installation files.

IBM WebSphere Real Time for AIX is provided in two types of package.

Installable packages
Installable packages configure your system. For example, the programs might set environment variables. Extracting this package provides the AIX installp filesets that you install with the **smitty** tool.

- `wrt-3.0-0.0-aix-<arch>-sdk-tar.gz`

The JRE is available only as an archive package.

Archive packages
These packages extract the files to your system, but do not perform any configuration. These are InstallAnywhere packages.

- `wrt-3.0-0.0-aix-<arch>-sdk-archive.bin`
- `wrt-3.0-0.0-aix-<arch>-jre-archive.bin`

Note: <arch> is your platform architecture; ppc_32 or ppc_64.

Before you begin, ensure that the AIX operating system is correctly configured, and the required patches are installed. Details can be found here: "Supported environments" on page 5. In particular, ensure that you have installed the required APARs for your system.

Installing from an installp package

After downloading the installation file you must extract the AIX filesets before installing WebSphere Real Time for AIX.

Before you begin

Ensure that you have downloaded the correct installation package specified in "Installation files." If you are accessing Passport Advantage®, this file might have a different name.

Procedure

These steps need to be performed one time only:

1. Extract the tar file from the installation package with the following command:
   ```
   gunzip <package>
   ```
   where `<package>` is the installable package `wrt-3.0-0.0-aix-<arch>-sdk-tar.gz`.
2. Extract the installp filesets from the tar file with the following command:
   ```
   tar xvf <tar_file>
   ```

© Copyright IBM Corp. 2003, 2014

where <tar_file> is the extracted tar file from step 1.

3. Use the AIX **installp** command to install WebSphere Real Time for AIX.
4. When the installation process is completed, follow the configuration steps in this section, starting with "Configuring user accounts" on page 13.

Relocating WebSphere Real Time for AIX

By default, the WebSphere Real Time for AIX SDK is installed in /usr/javawrt3[_64]/. To install in another directory, use the AIX relocation commands.

Delete any .toc files in the directory containing your installp images or PTFs before using the AIX relocation commands.

Commands

See the AIX man pages for reference information about the command-line options for these commands.

installp_r

Install the SDK:

```
installp_r -a -Y -R /<Install Path>/ -d '.' <fileset>
```

Remove the SDK:

```
installp_r -u  -R /<Install Path>/ <fileset>
```

lsusil List the user-defined installation paths.

```
lsusil
```

lslpp_r

Find details of installed products.

```
lslpp_r -R /<Install Path>/ -S [A|O]
```

rmusil Remove existing user-defined installation paths.

```
rmusil -R /<Install Path>/
```

Installing from an InstallAnywhere package

These packages provide an interactive program that guides you through the installation options. You can run the program as a graphical user interface, or from a system console.

About this task

The InstallAnywhere packages have a .bin file extension.

Procedure

- To install the package in an interactive way, complete an attended installation.
- To install the package without any additional user interaction, complete an unattended installation. You might choose this option if you want to install many systems.
- When the installation process is completed, follow the configuration steps in this section, such as settting path and classpath environment variables.

Results

Completing an attended installation

Install the product from an InstallAnywhere package, in an interactive way.

Before you begin

Check the following conditions before you begin the installation process:

- You must have a user ID with root authority.

Procedure

1. Download the installation package file to a temporary directory.
2. Change to the temporary directory.
3. Start the installation process by typing `./package` at a shell prompt, where *package* is the name of the package that you are installing.
4. Select a language from the list shown in the installer window, then click **Next**. The list of available languages is based on the locale setting for your system.
5. Read the license agreement, using the scroll bar to reach the end of the license text. To proceed with the installation you must accept the terms of the license agreement. To accept the terms, select the radio button, then click **OK**.

   Note: You cannot select the radio button to accept the license agreement until you have read to the end of the license text.
6. You are asked to choose the target directory for the installation. If you do not want to install into the default directory, click **Choose** to select an alternative directory, by using the browser window. When you have chosen the installation directory, click **Next** to continue.
7. You are asked to review the choices that you made. To change your selection, click **Previous**. If your choices are correct, click **Install** to proceed with installation.
8. When the installation process is complete, click **Done** to finish.

Completing an unattended installation

If you have more than one system to install, and you already know the installation options that you want to use, you might want to use the unattended installation process. You install once by using the attended installation process, then use the resulting response file to complete further installations without any additional user interaction.

Procedure

1. Create a response file by completing an attended installation. Use one of the following options:
   - Use the GUI and specify that the installation program creates a response file. The response file is called `installer.properties`, and is created in the installation directory.
   - Use the command line and append the -r option to the attended installation command, specifying the full path to the response file. For example:

     `./package -r /path/installer.properties`

   Example response file contents:

   ```
   INSTALLER_UI=silent
   USER_INSTALL_DIR=/my_directory
   ```

In this example, */my_directory* is the target installation directory that you chose for the installation.

2. Optional: If required, edit the response file to change options.

   Note: The packages have the following known issue: installations that use a response file use the default directory even if you change the directory in the response file. If a previous installation exists in the default directory, it is overwritten.

   If you are creating more than one response file, each with different installation options, specify a unique name for each response file, in the format *myfile*.properties.

3. Optional: Generate a log file. Because you are installing silently, no status messages are displayed at the end of the installation process. To generate a log file that contains the status of the installation, complete the following steps:
   a. Set the required system properties by using the following command.
      ```
      export _JAVA_OPTIONS="-Dlax.debug.level=3 -Dlax.debug.all=true"
      ```
   b. Set the following environment variable to send the log output to the console.
      ```
      export LAX_DEBUG=1
      ```
4. Start an unattended installation by running the package installer with the **-i** silent option, and the **-f** option to specify the response file. For example:
   ```
   ./package -i silent -f /path/installer.properties 1>console.txt 2>&1
   ./package -i silent -f /path/myfile.properties 1>console.txt 2>&1
   ```

   You can use a fully qualified path or relative path to the properties file. In these examples, the string `1>console.txt 2>&1` redirects installation process information from the stderr and stdout streams to the `console.txt` log file in the current directory. Review this log file if you think there was a problem with the installation.

   Note: If your installation directory contains multiple response files, the default response file, `installer.properties` is used.

Interrupted installation

If the package installer is unexpectedly stopped during installation, for example if you press Ctrl+C, the installation is corrupted and you cannot uninstall or reinstall the product. If you try to uninstall or reinstall you might see the message `Fatal Application Error`.

About this task

To solve this problem, delete files and reinstall, as described in the following steps.

Procedure

1. Delete the `/var/.com.zerog.registry.xml` registry file.
2. Delete the directory containing the installation, if it was created. For example /usr/javawrt3[_64]/.
3. Run the installation program again.

Known issues and limitations

The InstallAnywhere packages have some known issues and limitations.

- The installation package GUI does not support the Orca screen-reading program. You can use the unattended installation mode as an alternative to the GUI.
- If, after installation, you enter `./package` to start the program again, the program displays the following message:

  ```
  ENTER THE NUMBER OF THE DESIRED CHOICE, OR PRESS <ENTER> TO ACCEPT THE DEFAULT:
  ```

  If you press Enter to accept the default, the program does not respond. Type a number, then press Enter.
- If you install the package, then attempt to install again in a different mode, for example console or silent, you might see the following error message:

  ```
  Invocation of this Java Application has caused an InvocationTargetException.
  This application will now exit
  ```

  You should not see this message if you installed by using the GUI mode and are running the installation program again in console mode. .
- If you change the installation directory in a response file, and then run an unattended installation by using that response file, the installation program ignores the new installation directory and uses the default directory instead. If a previous installation exists in the default directory, it is overwritten.

Configuring user accounts

Important steps to configure AIX user accounts correctly on your system.

Procedure

This step must be completed one time only:

1. When the installation process is completed, you must change the user account to allow access to high-resolution timers. Run the following command as root user:

   ```
   chuser "capabilities=CAP_NUMA_ATTACH,CAP_PROPAGATE" <username>
   ```

   where *<username>* is the non-root AIX user account.

   Note: The user must log out and log back in for the change to take effect.

This step must be completed in every shell before starting Java:

1. Set the **AIXTHREAD_HRT** environment variable to true. This environment variable allows a process to use high-resolution timeouts with clock_nanosleep(). You must set this environment variable each time the process is started. On the command line, type:

   ```
   AIXTHREAD_HRT=true
   ```

   This setting can be added to a user's `.profile` so that it is set each time the user logs in. Add the following line to the user `.profile` file:

   ```
   export AIXTHREAD_HRT=true
   ```

Setting the path

Updating the **PATH** environment variable enables the operating system to find Java programs and utilities.

About this task

The **PATH** environment variable enables the operating system to find programs and utilities, such as **javac**, **java**, and **javadoc** tool from any current directory. Changing the path will override any existing Java launchers in your path.

To display the current value of your **PATH** environment variable, type the following command at a command prompt:

```
echo $PATH
```

To add the Java launchers to your path:

1. Edit the shell startup file in your home directory. The name of your startup file will depend on the shell you are using; for example:
   - The Korn shell startup file is **.kshrc**.
   - The C shell startup file is **.cshrc**.
   - The Bourne shell startup file is **.profile**.
   - The BASH shell startup file is **.bashrc**.

   Add the absolute paths to the **PATH** environment variable; for example:

   ```
   export PATH=/usr/javawrt3[_64]/jre/bin:/usr/javawrt3[_64]/bin:$PATH
   ```

   Note: The actual path name varies, depending on whether you used the default installation directory.
2. Log on again or run the updated shell script to activate the new **PATH** environment variable.

Results

After setting the path, you can run a tool by typing the tool command name at a command prompt from any directory. For example, to compile the file `Myfile.Java`, type:

```
javac Myfile.Java
```

Setting the classpath

The classpath tells the SDK tools, such as **java**, **javac**, and the **javadoc** tool, where to find the Java class libraries.

About this task

Set the classpath explicitly only for these reasons:

- You require a different library or class file, such as one that you develop, and it is not in the current directory.
- You change the location of the `bin` and `lib` directories and they no longer have the same parent directory.
- You plan to develop or run applications using different runtime environments on the same system.

To display the current value of your **CLASSPATH** environment variable, type the following command at a shell prompt:

```
echo $CLASSPATH
```

If you develop and run applications that use different runtime environments, including other versions that you have installed separately, you must set the **CLASSPATH** and **PATH** explicitly for each application. If you run multiple applications simultaneously and use different runtime environments, each application must run in its own shell.

Verifying the installation

Follow these steps to check that your installation was successful.

Before you begin

To help ensure that the verification process behaves consistently, first run these commands:

```
unset LIBPATH
unset CLASSPATH
unset JAVA_COMPILER
unset JAVA_HOME
export PATH=/usr/javawrt3[_64]/jre/bin:/usr/javawrt3[_64]/bin:$PATH
```

Procedure

Enter the following command:

```
java -Xgcpolicy:metronome -version
```

If the installation was successful, the following information is displayed:

```
java version "1.7.0"
WebSphere Real Time V3 (build pap3270-20110428_04)
IBM J9 VM (build 2.6, JRE 1.7.0 AIX ppc-32 20110427_81014 (JIT enabled, AOT enabled)
J9VM - R26_head_20110426_2022_B81001
JIT  - r11_20110426_19388
GC   - R26_head_20110426_1548_B80973
J9CL - 20110427_81014)
JCL - 20110427_03 based on Oracle 7b145
```

Dates, times, and specific build information might be different.

What to do next

When verification is complete, log on again and review any values that you might have assigned to these variables for possible conflicts.

Unless the `.hotjava` directory already exists, run the applet viewer to create a directory called `.hotjava` in your home directory. Enter the following command to confirm that the directory has been created:

```
ls -a ~
```

Chapter 5. Running IBM WebSphere Real Time for AIX applications

Important information to assist you when running real time applications.

-
-
- "Using the Metronome Garbage Collector"

Using the Metronome Garbage Collector

Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for AIX.

Controlling pause time

Metronome garbage collector (GC) pause time can be fine-tuned for each Java process.

By default, the Metronome GC pauses for 3 milliseconds in each individual pause, which is known as a quantum. A full garbage collection cycle requires many of these pauses, which are spread out to give the application enough time to run. You can change this maximum individual pause time value with the **-Xgc:targetPauseTime** option. For example, running with **-Xgc:targetPauseTime=20** causes the GC to operate with individual pauses that are no longer than 20 milliseconds.

The IBM Monitoring and Diagnostics Tools for Java - Garbage Collection and Memory Visualizer (GCMV) can be used to monitor the GC pause times for your application, as well as helping to diagnose and tune performance problems in your Java application. The tool parses and plots data from various types of log, including:

- Verbose garbage collection logs.
- Trace garbage collection logs, generated by using the **-Xtgc** parameter.
- Native memory logs, generated by using the **ps**, **svmon**, or **perfmon** system commands.

The graphs in this section are generated by GCMV, and show the affect of changing the target pause time on garbage collection cycles. Each graph plots the actual pause times between metronome garbage collection cycles (Y-axis) against the run time of an application (X-axis).

Note: GCMV supports an older verbose garbage collection format. If you want to analyze verbose GC output with GCMV, generate the output with the **-Xgc:verboseFormat=deprecated** option. For more information, see GC command-line options.

With the default target pause time set, the Verbose GC pause time graph shows that pause times are held around or below the 3 millisecond mark:

© Copyright IBM Corp. 2003, 2014

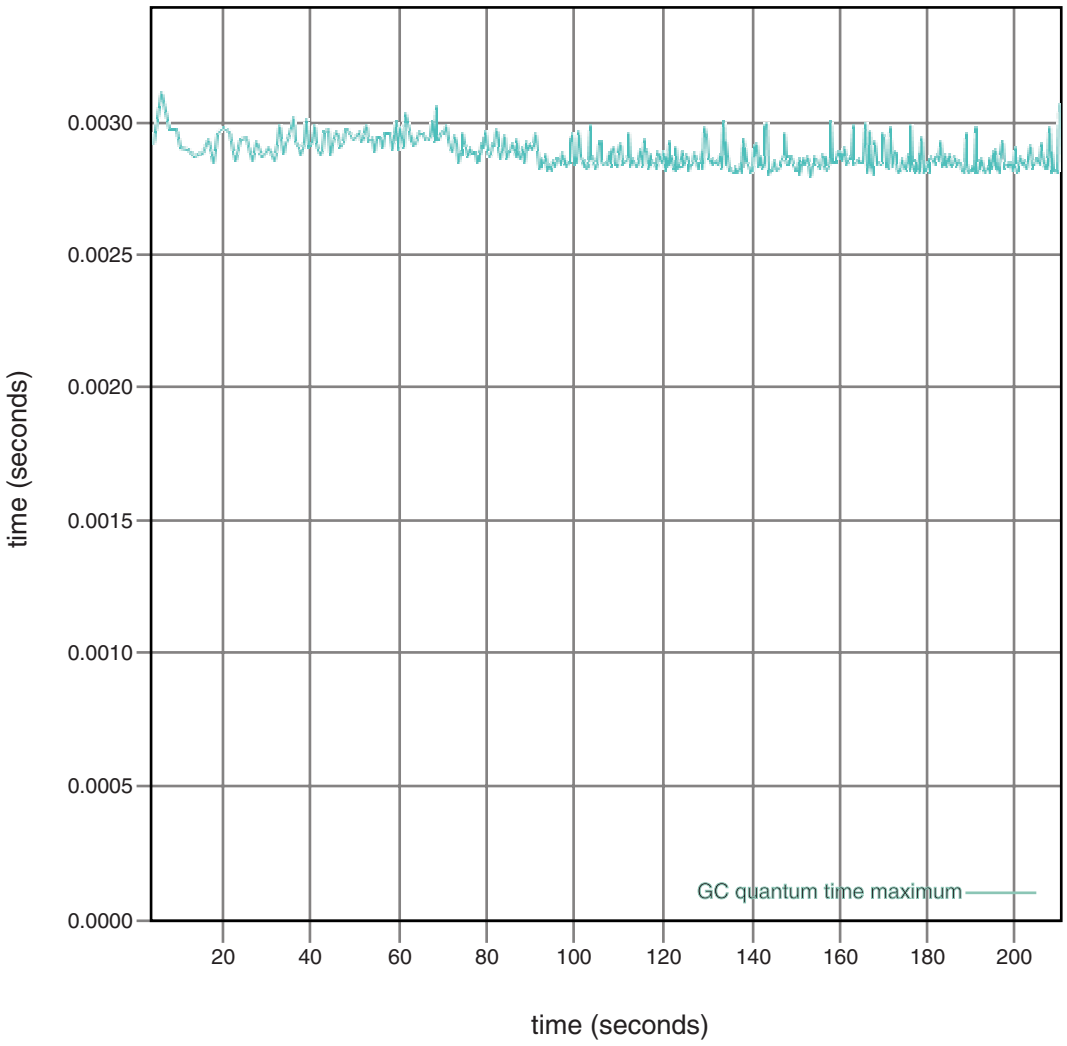

Figure 1. Actual garbage collection pause times when the target pause time is set to the default (3 milliseconds)

With a target pause time set at 6 milliseconds, the Verbose GC pause time graph shows that pause times are held around or below the 6 millisecond mark:

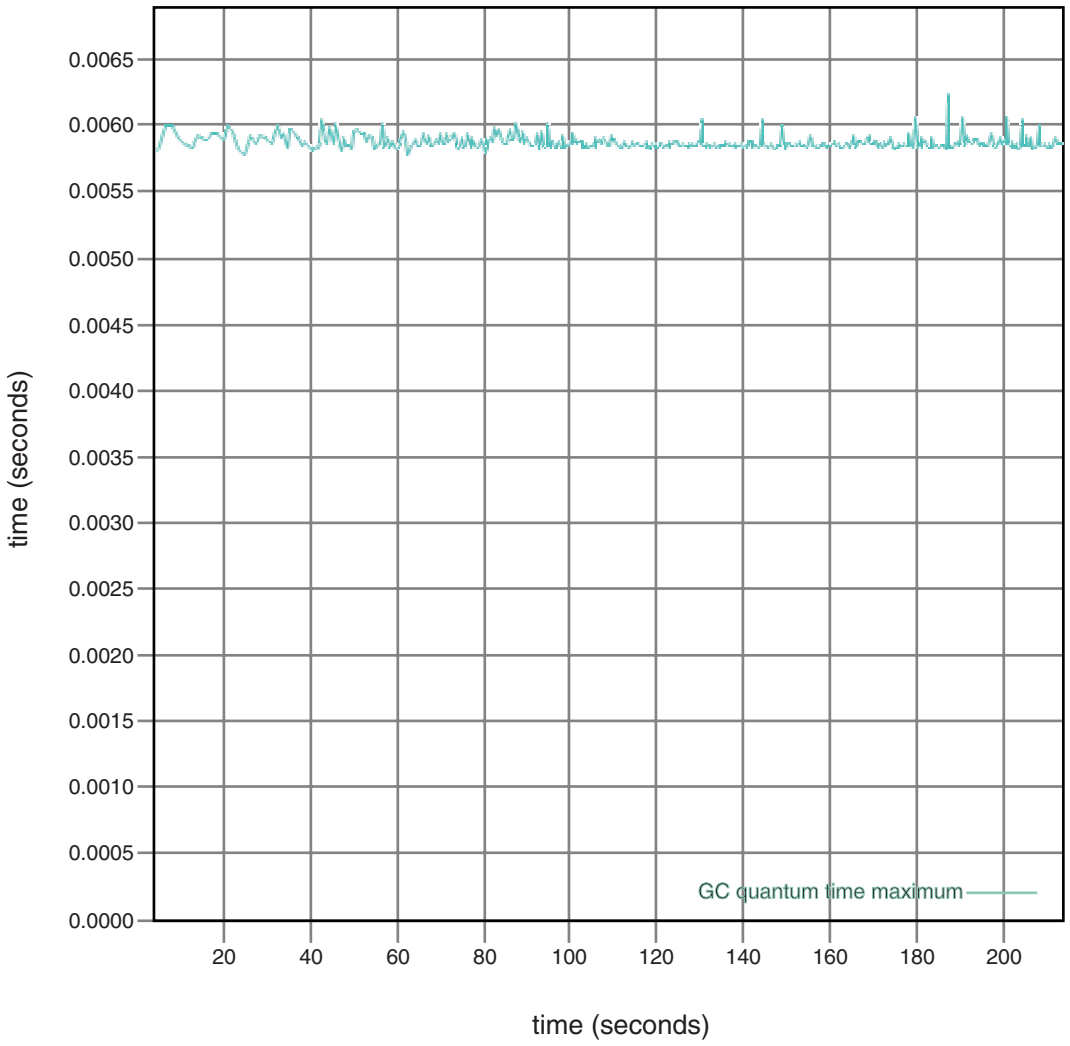

Figure 2. Actual pause times when the target pause time is set to 6 milliseconds

With a target pause time set at 10 milliseconds, the Verbose GC pause time graph shows that pause times are held around or below the 10 millisecond mark:

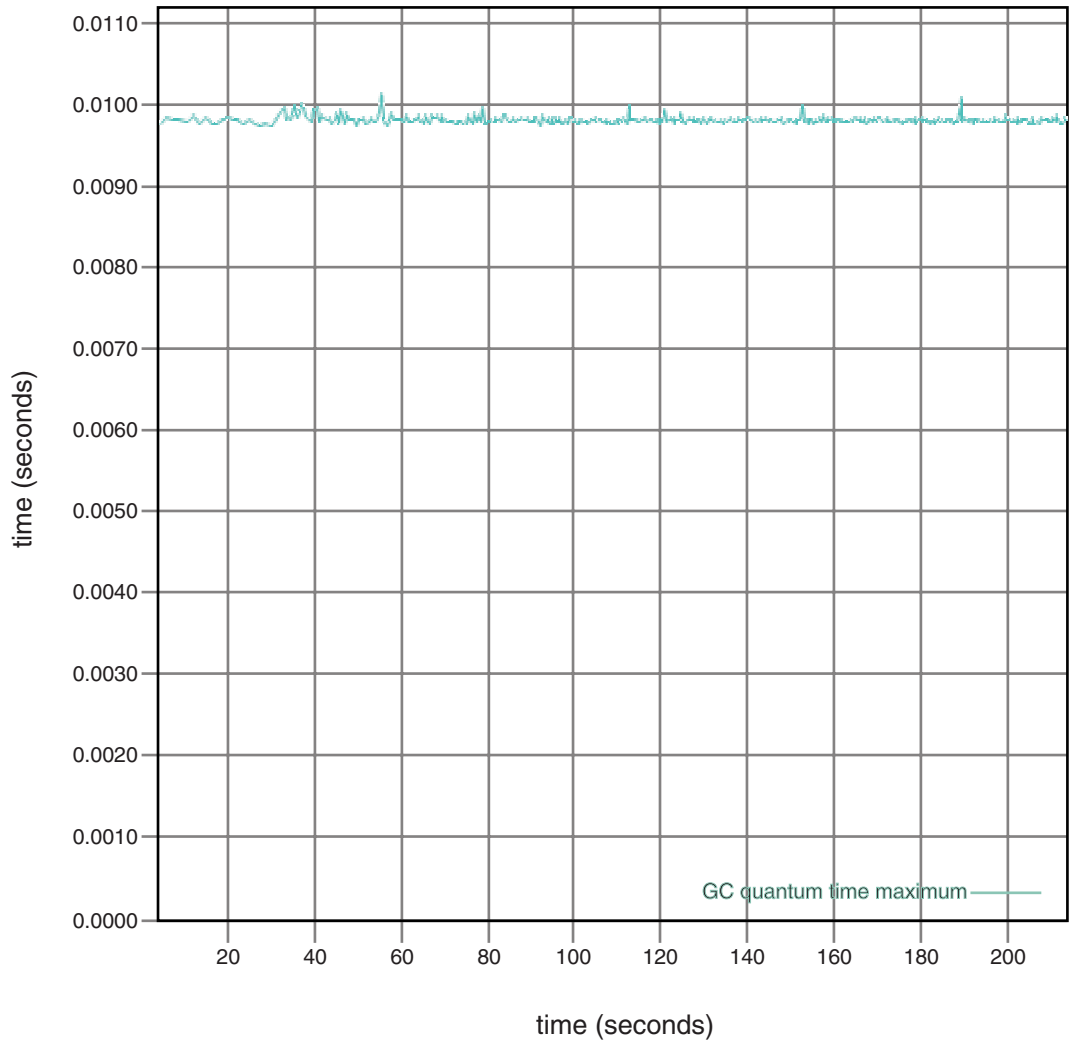

Figure 3. Actual pause times when the target pause time is set to 10 milliseconds

With a target pause time set at 15 milliseconds, the Verbose GC pause time graph shows that pause times are held around or below the 15 millisecond mark:

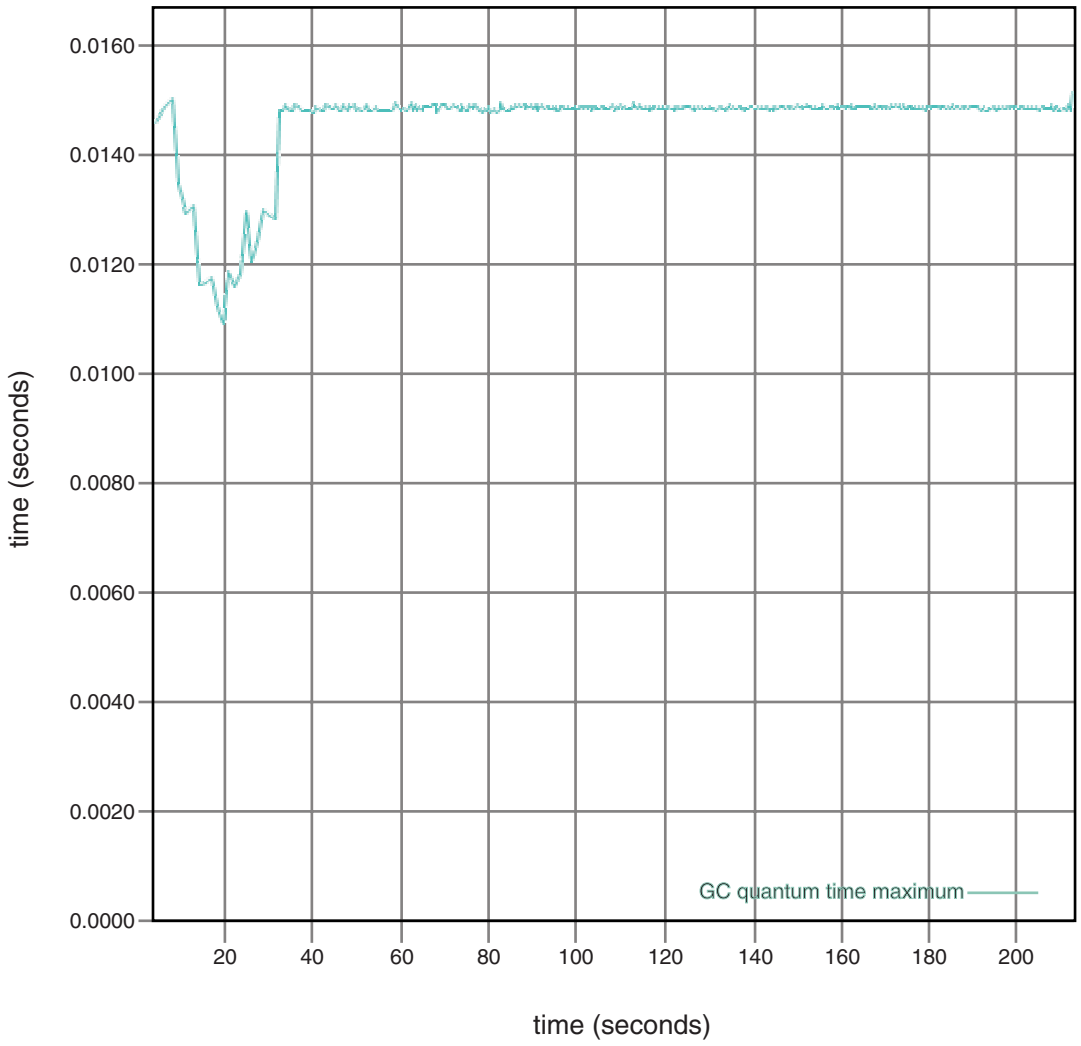

Figure 4. Actual pause times when the target pause time is set to 15 milliseconds

Controlling processor utilization

You can limit the amount of processing power available to the metronome garbage collector.

You can control garbage collection with the Metronome Garbage Collector using the **-Xgc:targetUtilization=N** option to limit the amount of CPU used by the Garbage Collector.

For example:

```
java -Xgcpolicy:metronome -Xgc:targetUtilization=80 yourApplication
```

The example specifies that your application runs for 80% in every 60 milliseconds. The remaining 20% of the time is used for garbage collection. The Metronome Garbage Collector guarantees utilization levels provided that it has been given sufficient resources. Garbage collection begins when the amount of free space in the heap falls below a dynamically determined threshold.

Metronome Garbage Collector limitations

This topic captures any known issues or limitations that affect the metronome GC policy.

Long pause times during garbage collection

Under rare circumstances, you might experience longer than expected pauses during garbage collection. During garbage collection, a root scanning process is used. The garbage collector walks the heap, starting at known live references. These references include:

- Live reference variables in the active thread call stacks.
- Static references.

To find all the live object references on an application thread's stack, the garbage collector scans all the stack frames in that thread's call stack. Each active thread stack is scanned in an uninterruptible step. Therefore the scan must take place within an individual GC pause.

The effect is that the system performance might be worse than expected if you have some threads with very deep stacks, because of extended garbage collection pauses at the beginning of a collection cycle.

Chapter 6. Developing applications

Important information about writing real time applications, including code samples.

The sample real-time hash map

WebSphere Real Time for AIX includes HashMap and HashSet implementations that provide more consistent performance for the put method than the standard HashMap in the IBM SDK for Java 7.

The standard java.util.HashMap that IBM provides works well for high throughput applications. It also helps with applications that know the maximum size their hash map needs to grow to. For applications that need a hash map that could grow to variable sizes, depending on usage, there is a potential performance problem with the standard hash map. The standard hash map provides good response times for adding new entries into the hash map using the put method. However, when the hash map fills up, a larger backing store must be allocated. This means that the entries in the current backing store must be migrated. If the hash map is large, the time to perform a put could also be large. For example, the operation could take several milliseconds.

WebSphere Real Time for AIX includes a sample real-time hash map. It provides the same functional interface as the standard java.util.HashMap, but enables much more consistent performance for the put method. Instead of creating a backing store and migrating all the entries when the hash map fills up, the sample hash map creates an additional backing store. The new backing store is chained to the other backing stores in the hash map. The chaining initially causes a slight performance reduction while the empty backing store is allocated and chained to the other backing stores. Once the backing hash map is updated, it is faster than having to migrate all the entries. A disadvantage of the real-time hash map is that the get, put and remove operations are slightly slower. The operations are slower because each look-up must to proceed through a set of backing hash maps instead of just one.

To try out the real-time hash map, add the `RTHashMap.jar` file to the start of your boot class path. If you installed WebSphere Real Time for AIX into the directory `$WRT_ROOT`, then add the following option to use the real-time hash map with your application, instead of the standard hash map:

```
-Xbootclasspath/p:$WRT_ROOT/demo/realtime/RTHashMap.jar
```

The source and class files for the real-time hash map implementation are included in the `demo/realtime/RTHashMap.jar` file. In addition, a real time java.util.LinkedHashMap and java.util.HashSet implementation are also provided.

© Copyright IBM Corp. 2003, 2014

Chapter 7. Performance

WebSphere Real Time for AIX is optimized for consistently short GC pauses rather than the highest throughput performance or smallest memory footprint.

Performance on certified hardware configurations

Certified systems have sufficient clock granularity and processor speed to support WebSphere Real Time for AIX performance goals. For example, a well-written application running on a system that is not overloaded, and with an adequate heap size, would normally experience GC pause times of about 3 milliseconds, and no more than 3.2 milliseconds. During GC cycles, an application with default environment settings is not paused for more than 30% of elapsed time during any sliding 60 millisecond window. The collective time spent in GC pauses over any 60 millisecond period normally totals about 18 milliseconds.

Reducing timing variability

The main sources of variability in a standard JVM are garbage collection pauses. In WebSphere Real Time for AIX, the potentially long pauses from standard Garbage Collector modes are avoided by using the Metronome Garbage Collector. See “Using the Metronome Garbage Collector” on page 17.

Class data sharing between JVMs

Class data sharing provides a transparent method of reducing memory footprint and improving JVM start time. To learn more on class data sharing see “Class data sharing between JVMs”

Compressed references

The Metronome GC supports both compressed and uncompressed references on 64-bit platforms. When using compressed references, the JVM stores all references to objects, classes, threads, and monitors as 32-bit values. Using compressed references improves the performance of many applications because objects are smaller, resulting in less frequent garbage collection and improved memory cache utilization.

Note: The heap size available for compressed references is limited to about 28 GB. For further information about compressed references, see http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.aix.70.doc/diag/understanding/mm_compressed_references.html.

Class data sharing between JVMs

Support for shared classes is the same when running with, or without, the **-Xrealtime** option.

You can share class data between Java Virtual Machines (JVMs) by storing it in a memory-mapped cache file on disk. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is deleted.

© Copyright IBM Corp. 2003, 2014

A shared cache can contain:

- Bootstrap classes
- Application classes
- Metadata that describes the classes
- Ahead-of-time (AOT) compiled code

Note: A real-time shared classes cache cannot be removed by a non real-time JVM.

Chapter 8. Security

This section contains important information about security.

Security considerations for the shared class cache

The shared class cache is designed for ease of cache management and usability, but the default security policy might not be appropriate.

When using the shared class cache, you must be aware of the default permissions for new files so that you can improve security by restricting access.

File	Default permissions
new shared caches	read permissions for group and other
`javasharedresources` directory	world read, write, and execute permission

You require write permission on both the cache file and the cache directory to destroy or grow a cache.

Changing the file permissions on the cache file

To limit access to a shared class cache, you can use the **chmod** command.

Change required	Command
Limit access to the user and group	`chmod 770 /tmp/javasharedresources`
Limit access to the user	`chmod 700 /tmp/javasharedresources`
Limit the user to read and write access only for a particular cache	`chmod 600 /tmp/javasharedresources/<file for shared cache>`
Limit the user and group to read and write access only for a particular cache	`chmod 660 /tmp/javasharedresources/<file for shared cache>`

Connecting to a cache that you do not have permission to access

If you try to connect to a cache that you do not have the appropriate access permissions for, you see an error message:

```
JVMSHRC226E Error opening shared class cache file
JVMSHRC220E Port layer error code = -302
JVMSHRC221E Platform error message: Permission denied
JVMJ9VM015W Initialization error for library j9shr25(11): JVMJ9VM009E J9VMDllMain
failed
Could not create the Java virtual machine.
```

© Copyright IBM Corp. 2003, 2014

Chapter 9. Troubleshooting and support

Troubleshooting and support for WebSphere Real Time for AIX

General problem determination methods

Problem determination helps you understand the kind of fault you have, and the appropriate course of action.

When you know what kind of problem you have, you might do one or more of the following tasks:

- Fix the problem.
- Find a good workaround.
- Collect the necessary data with which to generate a bug report to IBM.

AIX problem determination

This section describes problem determination on AIX.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems on AIX, covering:

- Setting up and checking your AIX environment
- General debugging techniques
- Diagnosing crashes
- Debugging hangs
- Understanding memory usage
- Debugging performance problems

You can find this information here: IBM SDK for Java 7 - AIX problem determination.

The following information is supplementary for IBM WebSphere Real Time for AIX

Setting up and checking your AIX environment

Check that your path statement is correctly set up. See "Setting the path" on page 13.

Debugging with the DBX plug-in

The plug-in for the AIX DBX debugger gives DBX users enhanced features when working on Java processes or core files generated by Java processes.

To enable the plug-in on 32-bit AIX, use the DBX command **pluginload**:

```
pluginload $JAVAHOME/jre/lib/ppc/softrealtime/libdbx_j9.so
```

On 64-bit AIX, use:

© Copyright IBM Corp. 2003, 2014

```
pluginload $JAVAHOME/jre/lib/ppc64/softrealtime/libdbx_j9.so
```

You can also set the **DBX_PLUGIN_PATH** environment variable to **$JAVAHOME/jre/lib/ppc[64]/softrealtime**. DBX automatically loads any plug-ins found in the specified path.

For more information about using the DBX plug-in, see DBX plug-in.

NLS problem determination

The JVM contains built-in support for different locales.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems with NLS, covering:

- Overview of fonts
- Font utilities
- Common NLS problems and possible causes

You can find this information here: IBM SDK for Java 7 - NLS problem determination.

ORB problem determination

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application. Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems with ORB, covering:

- Identifying an ORB problem
- Interpreting the stack trace
- Interpreting ORB traces
- Common problems
- IBM ORB service: Collecting data

You can find this information here: IBM SDK for Java 7 - ORB problem determination.

The following information is supplementary for IBM WebSphere Real Time for AIX.

IBM ORB service: collecting data

When collecting the Java version output for service, run the following command:

```
java -Xgcpolicy:metronome -version
```

Preliminary tests

When a problem occurs, the ORB might generate a `org.omg.CORBA.*` exception that includes:

- text to indicate the cause
- a minor code

- a completion status

Before you assume that the ORB is the cause of the problem, check these items:

- The scenario can be reproduced in a similar configuration.
- The JIT is disabled.
- No AOT compiled code is being used

Other actions include:

- Turn off additional processors.
- Turn off Simultaneous Multithreading (SMT) where possible.
- Eliminate memory dependencies with the client or server. The lack of physical memory can be the cause of slow performance, apparent hangs, or crashes. To remove these problems, ensure that you have a reasonable headroom of memory.
- Check physical network problems such as firewalls, communication links, routers, and DNS name servers. These are the major causes of CORBA COMM_FAILURE exceptions. As a test, ping your own workstation name.
- If the application is using a database such as DB2®, switch to the most reliable driver. For example, to isolate DB2 AppDriver, switch to Net Driver, which is slower and uses sockets, but is more reliable.

Troubleshooting OutOfMemory Errors

Dealing with OutOfMemoryError exceptions.

For general troubleshooting information on the Metronome Garbage Collector, see "Troubleshooting the Metronome Garbage Collector" on page 54.

Diagnosing OutOfMemoryErrors

Diagnosing OutOfMemoryError exceptions in Metronome Garbage Collector can be more complex than in a standard JVM because of the periodic nature of the garbage collector.

In general, a realtime application requires approximately 20% more heap space than a standard Java application.

By default, the JVM produces the following diagnostic output when an uncaught OutOfMemoryError occurs:

- A snap dump; see "Using dump agents" on page 37.
- A Heapdump; see "Using Heapdump" on page 44.
- A Javadump; see "Using Javadump" on page 39
- A system dump; see "Using system dumps and the dump viewer" on page 46.

The dump file names are given in the console output:

```
JVMDUMP006I Processing dump event "systhrow", detail "java/lang/OutOfMemoryError" - please wait.
JVMDUMP007I JVM Requesting Snap dump using 'Snap.20081017.104217.13161.0001.trc'
JVMDUMP010I Snap dump written to Snap.20081017.104217.13161.0001.trc
JVMDUMP007I JVM Requesting Heap dump using 'heapdump.20081017.104217.13161.0002.phd'
JVMDUMP010I Heap dump written to heapdump.20081017.104217.13161.0002.phd
JVMDUMP007I JVM Requesting Java dump using 'javacore.20081017.104217.13161.0003.txt'
JVMDUMP010I Java dump written to javacore.20081017.104217.13161.0003.txt
JVMDUMP013I Processed dump event "systhrow", detail "java/lang/OutOfMemoryError".
```

The Java backtrace shown on the console output, and also available in the Javadump, indicates where in the Java application the OutOfMemoryError occurred. The JVM memory management component issues a tracepoint that gives the size, class block address, and memory space name of the failing allocation. This tracepoint can be found in the snap dump:

```
<< lines omitted... >>
09:42:17.563258000 *0xf2888e00       j9mm.101  Event       J9AllocateIndexableObject() returning NULL! 80
bytes requested for object of class 0xf1632d80 from memory space 'Metronome' id=0xf288b584
```

The tracepoint ID and data fields might vary from that shown, depending on the type of object being allocated. In this example, the tracepoint shows that the allocation failure occurred when the application attempted to allocate a 33.6 MB object of type `class 0x81312d8` in the `Metronome heap`, memory segment `id=0x809c5f0`.

You can determine which memory area is affected by looking at the memory management information in the Javadump:

```
NULL           ------------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
NULL
1STMEMTYPE     Object Memory
NULL           region     start      end        size       name
1STHEAP        0xF288B584 0xF2A1C000 0xF6A1C000 0x04000000 Default
NULL
1STMEMUSAGE    Total memory available: 67108864 (0x04000000)
1STMEMUSAGE    Total memory in use:    66676824 (0x03F96858)
1STMEMUSAGE    Total memory free:      00432040 (0x000697A8)

<< lines removed for clarity >>
```

You can determine the type of object being allocated by looking at the classes section of the Javadump:

```
NULL           ------------------------------------------------------------------
0SECTION       CLASSES subcomponent dump routine
NULL           ================================
<< lines omitted... >>
1CLTEXTCLLOD    ClassLoader loaded classes
2CLTEXTCLLOAD    Loader *System*(0xF182BB80)
<< lines omitted... >>
3CLTEXTCLASS      [C(0xF1632D80)
```

Information in the Javadump confirms that the attempted allocation was for a character array, in the normal heap (`ID=0xF288B584`) and that the total allocated size of the heap, indicated by the appropriate `1STHEAP` line, is 67108864 decimal bytes or 0x04000000 hex bytes, or 64 MB.

In this example, the failing allocation is large in relation to the total heap size. If your application is expected to create 33 MB objects, the next step is to increase the size of the heap, using the **-Xmx** option.

It is more common for the failing allocation to be small in relation to total heap size. This is because of previous allocations filling up the heap. In these cases, the next step is to use the Heapdump to investigate the amount of memory allocated to existing objects.

The Heapdump is a compressed binary file containing a list of all objects with their object class, size, and references. Analyze the Heapdump using the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer tool, which is available for download from the IBM Support Assistant (ISA).

Using MDD4J, you can load a Heapdump and locate tree structures of objects that are suspected of consuming large amounts of heap space. The tool provides various views for objects on the heap. For example, MDD4J can show a view that details likely leak suspects, and gives the top five objects and packages contributing to the heap size. Selecting the tree view gives further information about the nature of the leaking container object.

How the IBM JVM manages memory

The IBM JVM requires memory for several different components, including memory regions for classes, compiled code, Java objects, Java stacks, and JNI stacks. Some of these memory regions must be in contiguous memory. Other memory regions can be segmented into smaller memory regions and linked together.

Dynamically loaded classes and compiled code are stored in segmented memory regions for dynamically loaded classes. Classes are further subdivided into writable memory regions (RAM classes) and read-only memory regions (ROM classes). At run time, ROM classes and AOT code from the class cache are memory mapped, but not loaded, into a contiguous memory region on application startup. As classes are referenced by the application, classes and compiled code in the class cache are mapped into storage. The ROM component of the class is shared between multiple processes referencing this class. The RAM component of the class is created in the segmented memory regions for dynamically loaded classes when the class is first referenced by the JVM. AOT-compiled code for the methods of a class in the class cache are copied into an executable dynamic code memory region, because this code is not shared by processes. Classes that are not loaded from the class cache are similar to cached classes, except that the ROM class information is created in segmented memory regions for dynamically loaded classes. Dynamically generated code is stored in the same dynamic code memory regions that hold AOT code for cached classes.

The stack for each Java thread can span a segmented memory region. The JNI stack for each thread occupies a contiguous memory region.

To determine how your JVM is configured, run with the **-verbose:sizes** option. This option prints out information about memory regions where you can manage the size. For memory regions that are not contiguous, an increment is printed describing how much memory is acquired every time the region needs to grow.

Here is example output using the **-Xrealtime -verbose:sizes** options:

```
-Xmca32K                    RAM class segment increment
-Xmco128K                   ROM class segment increment
-Xms64M                     initial memory size

-Xmx64M                     memory maximum
-Xmso256K                   operating system thread stack size
-Xiss2K                     java thread stack initial size
-Xssi16K                    java thread stack increment
-Xss256K                    java thread stack maximum size
```

This example indicates that the RAM class segment is initially 0, but grows by 32 KB blocks as required. The ROM class segment is initially 0, and grows by 128 KB

blocks as required. You can use the **-Xmca** and **-Xmco** options to control these sizes. RAM class and ROM class segments grow as required, so you will not typically need to change these options.

Use the **-Xshareclasses** option to determine how large your memory mapped region will be if you use the class cache. Here is a sample of the output from the command `java -Xgcpolicy:metronome -Xshareclasses:printStats`.

```
Current statistics for cache "sharedcc_j9build":

shared memory ID = 103809029

base address = 0xC000DBB8
end address = 0xC1000000
allocation pointer = 0xC00F3340

cache size = 16776892
free bytes = 15536560
ROMClass bytes = 1161532
AOT bytes = 0
Data bytes = 56244
Metadata bytes = 22556
Metadata % used = 1%

# ROMClasses = 365
# AOT Methods = 0
# Classpaths = 1
# URLs = 0
# Tokens = 0
# Stale classes = 0
% Stale classes = 0%

Cache is 7% full
```

```
Current statistics for cache "sharedcc_j9build":

shared memory ID = 48234504

base address = 0x0700000002000DC0C
end address = 0x0700000021000000
allocation pointer = 0x07000000200F3394

cache size = 16776808
free bytes = 15267168
ROMClass bytes = 1412468
AOT bytes = 17728
Data bytes = 56504
Metadata bytes = 22940
Metadata % used = 1%

# ROMClasses = 365
# AOT Methods = 4
# Classpaths = 1
# URLs = 0
# Tokens = 0
# Stale classes = 0
% Stale classes = 0%

Cache is 8% full
```

At run time, approximately 3 MB of AOT bytes and metadata bytes are copied into the dynamic code segmented region, as the classes are referenced. The data bytes are copied into the RAM class segmented region, as the classes are referenced.

Using diagnostic tools

There are a number of diagnostic tools that are available to help diagnose problems with the IBM WebSphere Real Time for AIX JVM.

The IBM SDK for Java 7 provides a number of diagnostic tools that can be used to diagnose problems with the IBM WebSphere Real Time for AIX JVM. This section introduces the tools that are available, and provides links to further information about using the tools.

There is an important point to remember when using the SDK diagnostic tools. When you invoke the real time JVM, you use the following option:

```
java  -Xgcpolicy:metronome
```

This option must be used when running diagnostic tools for the real time JVM. For example, to show the registered dump agents for the IBM WebSphere Real Time for AIX JVM, type:

```
java -Xgcpolicy:metronome -Xdump:what
```

Any further differences in using these tools with IBM WebSphere Real Time for AIX is provided here as supplementary information, together with sample output to assist you with diagnosis.

For a summary of the diagnostic information that is generated by the IBM SDK for Java 7, see Summary of diagnostic information.

Using the IBM Monitoring and Diagnostic Tools for Java

IBM provides tooling and documentation to help you understand, monitor, and diagnose problems with applications using the IBM JRE.

The following tools are available:

- Health Center
- Garbage Collection and Memory Visualizer
- Interactive Diagnostic Data Explorer
- Memory Analyzer

Garbage Collection and Memory Visualizer

Garbage Collection and Memory Visualizer (GCMV) helps you understand memory use, garbage collection behavior, and performance of Java applications.

GCMV parses and plots data from various types of log, including the following types:

- Verbose garbage collection logs.
- Trace garbage collection logs, generated by using the -Xtgc parameter.
- Native memory logs, generated by using the ps, **svmon**, or **perfmon** system commands.

The tool helps to diagnose problems such as memory leaks, analyze data in various visual formats, and provides tuning recommendations.

GCMV is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/gcmv/.

Further information about GCMV is available in an IBM Information Center.

Health Center

Health Center is a diagnostic tool for monitoring the status of a running Java Virtual Machine (JVM).

The tool is provided in two parts:

- The Health Center agent that collects data from a running application.
- An Eclipse-based client that connects to the agent. The client interprets the data and provides recommendations to improve the performance of the monitored application.

Health Center is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/.

Further information about Health Center is available in an IBM Information Center.

Interactive Diagnostic Data Explorer

Interactive Diagnostic Data Explorer (IDDE) is a GUI-based alternative to the dump viewer (**jdmpview** command). IDDE provides the same functionality as the dump viewer, but with extra support such as the ability to save command output.

Use IDDE to more easily explore and examine dump files that are produced by the JVM. Within IDDE, you enter commands in an investigation log, to explore the dump file. The support that is provided by the investigation log includes the following items:

- Command assistance
- Auto-completion of text, and some parameters such as class names
- The ability to save commands and output, which you can then send to other people
- Highlighted text and flagging of issues
- The ability to add your own comments
- Support for using the Memory Analyzer from within IDDE

IDDE is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see IDDE overview on developerWorks®.

Further information about IDDE is available in an IBM Information Center.

Memory Analyzer

Memory Analyzer helps you analyze Java heaps using operating system level dumps and Portable Heap Dumps (PHD).

This tool can analyze dumps that contain millions of objects, providing the following information:

- The retained sizes of objects.
- Processes that are preventing the Garbage Collector from collecting objects.
- A report to automatically extract leak suspects.

This tool is based on the Eclipse Memory Analyzer (MAT) project, and uses the IBM Diagnostic Tool Framework for Java (DTFJ) feature to enable the processing of dumps from IBM JVMs.

Memory Analyzer is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/.

Further information about Memory Analyzer is available in an IBM Information Center.

Using dump agents

Dump agents are set up during JVM initialization. They enable you to use events occurring in the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate dumps or to start an external tool.

The IBM SDK for Java V7 User guide contains useful guidance on dump agents, covering:

- Using the **-Xdump** option
- Dump agents
- Dump events
- Advanced control of dump agents
- Dump agent tokens
- Default dump agents
- Removing dump agents
- Dump agent environment variables
- Signal mappings
- Dump agent default locations

You can find this information here: IBM SDK for Java 7 - Using dump agents.

Supplementary information for IBM WebSphere Real Time for AIX is provided here:

Dump events

Dump agents are triggered by events occurring during JVM operation. For IBM WebSphere Real Time for AIX, the default value for the slow event is 5 milliseconds.

Some events can be filtered to improve the relevance of the output. See “filter option” on page 38 for more information.

Note: The unload and expand events currently do not occur in WebSphere Real Time. Classes are in immortal memory and cannot be unloaded.

Note: The gpf and abort events cannot trigger a heap dump, prepare the heap (request=prepwalk), or compact the heap (request=compact).

The following table shows events available as dump agent triggers:

Event	Triggered when...	Filter operation
gpf	A General Protection Fault (GPF) occurs.	
user	The JVM receives the SIGQUIT signal from the operating system.	
abort	The JVM receives the SIGABRT signal from the operating system.	

Event	Triggered when...	Filter operation
vmstart	The virtual machine is started.	
vmstop	The virtual machine stops.	Filters on exit code; for example, **filter=#129..#192#-42#255**
load	A class is loaded.	Filters on class name; for example, **filter=java/lang/String**
unload	A class is unloaded.	
throw	An exception is thrown.	Filters on exception class name; for example, **filter=java/lang/OutOfMem***
catch	An exception is caught.	Filters on exception class name; for example, **filter=*Memory***
uncaught	A Java exception is not caught by the application.	Filters on exception class name; for example, **filter=*MemoryError**
systhrow	A Java exception is about to be thrown by the JVM. This is different from the 'throw' event because it is only triggered for error conditions detected internally in the JVM.	Filters on exception class name; for example, **filter=java/lang/OutOfMem***
thrstart	A new thread is started.	
blocked	A thread becomes blocked.	
thrstop	A thread stops.	
fullgc	A garbage collection cycle is started.	
slow	A thread takes longer than 5ms to respond to an internal JVM request.	Changes the time taken for an event to be considered slow; for example, **filter=#300ms** will trigger when a thread takes longer than 300ms to respond to an internal JVM request.
allocation	A Java object is allocated with a size matching the given filter specification	Filters on object size; a filter must be supplied. For example, **filter=#5m** will trigger on objects larger than 5 Mb. Ranges are also supported; for example, **filter=#256k..512k** will trigger on objects between 256 Kb and 512 Kb in size.
traceassert	An internal error occurs in the JVM	Not applicable.
corruptcache	The JVM finds that the shared class cache is corrupt.	Not applicable.

filter option

Some JVM events occur thousands of times during the lifetime of an application. Dump agents can use filters and ranges to avoid excessive dumps being produced.

Wildcards

You can use a wildcard in your exception event filter by placing an asterisk only at the beginning or end of the filter. The following command does not work because the second asterisk is not at the end:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

In order to make this filter work, it must be changed to:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

Class loading and exception events

You can filter class loading (load) and exception (throw, catch, uncaught, systhrow) events by Java class name:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

You can filter throw, uncaught, and systhrow exception events by Java method name:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.
throwingMethodName[#stackFrameOffset]]
```

Optional portions are shown in brackets.

You can filter the catch exception events by Java method name:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.
catchingMethodName]
```

Optional portions are shown in square brackets.

vmstop event

You can filter the JVM shut down event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

slow event

You can filter the slow event to change the time threshold from the default of 5ms:

```
-Xdump:java:events=slow,filter=#300ms
```

You cannot set the filter to a time lower than the default time.

Other events

If you apply a filter to an event that does not support filtering, the filter is ignored.

Using Javadump

Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

The IBM SDK for Java V7 User guide contains useful guidance on Javadumps, covering:

- Enabling a Javadump
- Triggering a Javadump
- Interpreting a Javadump
- Environment variables and Javadump

You can find this information here: IBM SDK for Java 7 - Using Javadump.

Supplementary information and sample output for IBM WebSphere Real Time for AIX is provided in the following topics.

Storage Management (MEMINFO)

The MEMINFO section provides information about the Memory Manager, including heap, immortal, and scoped memory areas.

The MEMINFO section of a Javadump shows information about the Memory Manager. See Using the Metronome Garbage Collector for details about how the memory manager component works.

This part of the Javadump provides various storage management values, including:

- amount of free memory
- amount of used memory
- current size of the heap
- current size of immortal memory areas
- current size of scoped memory areas

This section also contains garbage collection history data. The data is shown as a sequence of tracepoints, each with a timestamp, ordered with the most recent tracepoint first.

Javadumps produced by the standard JVM contain a "GC History" section. This information is not contained in Javadumps produced when using the real-time JVM. Use the **-verbose:gc** option or the JVM snap trace to obtain information about GC behavior. See "Using verbose:gc information" on page 54 and the dump agents section of the IBM SDK for Java V7 User guide for more details.

In a Javadump, segments are blocks of memory allocated by the Java run time for tasks that use large amounts of memory. Example tasks are:

- maintaining JIT caches
- storing Java classes

The Java runtime environment also allocates other native memory, which is not listed in the MEMINFO section. The total memory used by Java runtime segments does not necessarily represent the complete memory footprint of the Java run time. A Java runtime segment consists of the segment data structure, and an associated block of native memory.

The following example shows some typical output. All the values are provided as hexadecimal values. The column headings in the MEMINFO section have the following meanings:

- Object memory section (HEAPTYPE):

  id The id of the space or region.

  start The start address of this region of the heap.

  end The end address of this region of the heap.

  size The size of this region of the heap.

  space/region
  For a line that contains only an id and a name, this column shows the name of the memory space. Otherwise the column shows the name of the memory space, followed by the name of a particular region that is contained within that memory space.

- Internal memory section (SEGTYPE), including class memory, JIT code cache, and JIT data cache:

segment
The address of the segment control data structure.

start The start address of the native memory segment.

alloc The current allocation address within the native memory segment.

end The end address of the native memory segment.

type An internal bit field describing the characteristics of the native memory segment.

size The size of the native memory segment.

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
NULL
1STHEAPTYPE    Object Memory
NULL           id                 start              end                size               space/region
1STHEAPSPACE   0x0000010010FE8FF0         --                 --                 --         Metronome
1STHEAPREGION  0x000001001029D460 0x0000000040000000 0x0000000040010000 0x0000000000010000 Metronome/Small Region
1STHEAPREGION  0x000001001029D640 0x0000000040010000 0x0000000040020000 0x0000000000010000 Metronome/Small Region
1STHEAPREGION  0x000001001029D820 0x0000000040020000 0x0000000040030000 0x0000000000010000 Metronome/Small Region
....
1STHEAPREGION  0x000001001029F080 0x00000000400F0000 0x0000000040100000 0x0000000000010000 Metronome/Arraylet Region
1STHEAPREGION  0x000001001029F260 0x0000000040100000 0x0000000040110000 0x0000000000010000 Metronome/Arraylet Region
....
NULL
1STHEAPTOTAL   Total memory:           536870912 (0x0000000020000000)
1STHEAPINUSE   Total memory in use:    534955160 (0x000000001FE2C498)
1STHEAPFREE    Total memory free:        1915752 (0x00000000001D3B68)
NULL
1STSEGTYPE     Internal Memory
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x0000010012471558 0x0000010012D63870 0x0000010012D63870 0x0000010012D73870 0x01000040 0x0000000000010000
1STSEGMENT     0x0000010012471318 0x0000010012D33750 0x0000010012D33750 0x0000010012D43750 0x01000040 0x0000000000010000
1STSEGMENT     0x0000010012498B98 0x00000100125A3150 0x00000100125A3150 0x00000100125B3150 0x01000040 0x0000000000010000
....
NULL
1STSEGTOTAL    Total memory:          1784400 (0x00000000001B3A50)
1STSEGINUSE    Total memory in use:         0 (0x0000000000000000)
1STSEGFREE     Total memory free:     1784400 (0x00000000001B3A50)
NULL
1STSEGTYPE     Class Memory
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x000001001247F358 0x00000000601007F0 0x00000000601087F0 0x00000000601087F0 0x00010040 0x0000000000008000
1STSEGMENT     0x000001001247F298 0x0000010012BE0AD0 0x0000010012BE0E28 0x0000010012C00AD0 0x00020040 0x0000000000020000
1STSEGMENT     0x000001001247F1D8 0x00000000600F87A0 0x00000000601007A0 0x00000000601007A0 0x00010040 0x0000000000008000
....
NULL
1STSEGTOTAL    Total memory:          2329244 (0x0000000000238A9C)
1STSEGINUSE    Total memory in use:   2160420 (0x000000000020F724)
1STSEGFREE     Total memory free:      168824 (0x0000000000029378)
NULL
1STSEGTYPE     JIT Code Cache
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x00000100117B3FD8 0x00000100117B4270 0x00000100117D2878 0x00000100119B4270 0x00000068 0x0000000000200000
1STSEGMENT     0x000001001113BBB8 0x00000100115A4210 0x00000100115BA708 0x00000100117A4210 0x00000068 0x0000000000200000
1STSEGMENT     0x000001001113BAF8 0x0000010011394490 0x00000100113AA988 0x0000010011594490 0x00000068 0x0000000000200000
1STSEGMENT     0x000001001113BA38 0x0000010011184710 0x000001001119AC08 0x0000010011384710 0x00000068 0x0000000000200000
NULL
1STSEGTOTAL    Total memory:          8388608 (0x0000000000800000)
1STSEGINUSE    Total memory in use:    398576 (0x00000000000614F0)
1STSEGFREE     Total memory free:     7990032 (0x000000000079EB10)
NULL
1STSEGTYPE     JIT Data Cache
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x000001001113BF98 0x00000100119C4210 0x00000100119D7E40 0x0000010011BC4210 0x00000048 0x0000000000200000
NULL
1STSEGTOTAL    Total memory:          2097152 (0x0000000000200000)
1STSEGINUSE    Total memory in use:     80944 (0x0000000000013C30)
1STSEGFREE     Total memory free:     2016208 (0x00000000001EC3D0)
NULL
1STGCHTYPE     GC History
3STHSTTYPE     15:18:36:229787192 GMT j9mm.101 -   J9AllocateIndexableObject() returning NULL! 268451872 bytes requested for object
   of class 0000000060015600 from memory space 'Metronome' id=0000010010FE8FF0
3STHSTTYPE     15:18:36:216005016 GMT j9mm.468 -   Cycle End: type 1 approximateFreeMemorySize 2047080
3STHSTTYPE     15:18:36:216002364 GMT j9mm.475 -   GlobalGC end: workstackoverflow=0 overflowcount=0 memory=2047080/536870912
```

```
3STHSTTYPE     15:18:36:216000829 GMT j9mm.51 -   SystemGC end: newspace=0/0 oldspace=2047080/536870912 loa=0/0
3STHSTTYPE     15:18:36:215983093 GMT j9mm.57 -   Sweep end
3STHSTTYPE     15:18:36:209832984 GMT j9mm.56 -   Sweep start
3STHSTTYPE     15:18:36:209831045 GMT j9mm.94 -   Class unloading end: classloadersunloaded=0 classesunloaded=0
3STHSTTYPE     15:18:36:209822935 GMT j9mm.60 -   Class unloading start
3STHSTTYPE     15:18:36:209822201 GMT j9mm.55 -   Mark end
3STHSTTYPE     15:18:36:207800094 GMT j9mm.54 -   Mark start
3STHSTTYPE     15:18:36:207795841 GMT j9mm.474 -   GlobalGC start: globalcount=4
3STHSTTYPE     15:18:36:207794287 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.000 meanexclusiveaccessms=0.000 threads=0
   lastthreadtid=0x0000000000000000 beatenbyotherthread=0
3STHSTTYPE     15:18:36:207793480 GMT j9mm.131 -   SystemGC start: newspace=0/0 oldspace=2047080/536870912 loa=0/0
3STHSTTYPE     15:18:36:207790037 GMT j9mm.467 -   Cycle Start: type 1 approximateFreeMemorySize 2047080
3STHSTTYPE     15:18:36:204710301 GMT j9mm.468 -   Cycle End: type 1 approximateFreeMemorySize 2047080
3STHSTTYPE     15:18:36:204707992 GMT j9mm.475 -   GlobalGC end: workstackoverflow=0 overflowcount=0 memory=2047080/536870912
3STHSTTYPE     15:18:36:204706764 GMT j9mm.51 -   SystemGC end: newspace=0/0 oldspace=2047080/536870912 loa=0/0
....
NULL           ------------------------------------------------------------------------
```

Threads and stack trace (THREADS)

For the application programmer, one of the most useful pieces of a Java dump is the THREADS section. This section shows a list of Java threads, native threads, and stack traces.

A Java thread is implemented by a native thread of the operating system. Each thread is represented by a set of lines such as:

```
"main" J9VMThread:0x41D11D00, j9thread_t:0x003C65D8, java/lang/Thread:0x40BD6070, state:CW, prio=5
 (native thread ID:0xA98, native priority:0x5, native policy:UNKNOWN)
 Java callstack:
 at java/lang/Thread.sleep(Native Method)
 at java/lang/Thread.sleep(Thread.java:862)
 at mySleep.main(mySleep.java:31)
```

The properties on the first line are the thread name, addresses of the JVM thread structures and of the Java thread object, thread state, and Java thread priority. The properties on the second line are the native operating system thread ID, native operating system thread priority and native operating system scheduling policy.

Thread names are visible in three ways:

- Listed in javacore files. Not all threads are listed in javacore files.
- When listing threads from the operating system with the **ps** command.
- When using the java.lang.Thread.getName() method.

The following table provides information about IBM WebSphere Real Time for AIX thread names.

Table 2. Thread names in IBM WebSphere Real Time for AIX

Detail of thread	Thread name
An internal JVM thread used by the garbage collection module to dispatch the finalization of objects by secondary threads.	`Finalizer master`
The alarm thread used by the garbage collector.	`GC Alarm`
The slave threads used for garbage collection.	`GC Slave`
An internal JVM thread used by the just-in-time compiler module to sample the usage of methods in the application.	`IProfiler`

Table 2. Thread names in IBM WebSphere Real Time for AIX (continued)

Detail of thread	Thread name
A thread used by the VM to manage signals received by the application, whether externally or internally generated.	`Signal Reporter`
An internal JVM thread used to compile Java code.	`JIT Compilation Thread`
An internal JVM thread used to allow JVMTI agents to attach to a running JVM.	`Attach API wait loop`

The Java thread priority is mapped to an operating system priority value in a platform-dependent manner. A large value for the Java thread priority means that the thread has a high priority. In other words, the thread runs more frequently than lower priority threads.

The values of state can be:

- R - Runnable - the thread is able to run when given the chance.
- CW - Condition Wait - the thread is waiting. For example, because:
  - A sleep() call is made
  - The thread has been blocked for I/O
  - A wait() method is called to wait on a monitor being notified
  - The thread is synchronizing with another thread with a join() call
- S – Suspended – the thread has been suspended by another thread.
- Z – Zombie – the thread has been killed.
- P – Parked – the thread has been parked by the new concurrency API (java.util.concurrent).
- B – Blocked – the thread is waiting to obtain a lock that something else currently owns.

If a thread is parked or blocked, the output contains a line for that thread, beginning with `3XMTHREADBLOCK`, listing the resource that the thread is waiting for and, if possible, the thread that currently owns that resource. For more information see the topic on blocked threads in the IBM SDK for Java V7 User guide.

When you initiate a Javadump to obtain diagnostic information, the JVM quiesces Java threads before producing the javacore. A preparation state of `exclusive_vm_access` is shown in the `1TIPREPSTATE` line of the `TITLE` section.

```
1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)
```

Threads that were running Java code when the javacore was triggered are in CW (Condition Wait) state.

```
3XMTHREADINFO      "main" J9VMThread:0x41481900, j9thread_t:0x002A54A4, java/lang/Thread:0x004316B8,
state:CW, prio=5
3XMTHREADINFO1            (native thread ID:0x904, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO3           Java callstack:
4XESTACKTRACE                at java/lang/String.getChars(String.java:667)
4XESTACKTRACE                at java/lang/StringBuilder.append(StringBuilder.java:207)
```

The javacore `LOCKS` section shows that these threads are waiting on an internal JVM lock.

```
2LKREGMON          Thread public flags mutex lock (0x002A5234): <unowned>
3LKNOTIFYQ            Waiting to be notified:
3LKWAITNOTIFY            "main" (0x41481900)
```

Using Heapdump

The term Heapdump describes the IBM Virtual Machine for Java mechanism that generates a dump of all the live objects that are on the Java heap; that is, those that are being used by the running Java application.

The IBM SDK for Java V7 User guide contains useful guidance on Heapdumps, covering:

- Getting Heapdumps
- Tools for processing Heapdumps
- Using **-Xverbose:gc** to obtain heap information
- Environment variables and Heapdump
- Text (classic) Heapdump file format
- Portable Heap Dump (PHD) file format

You can find this information here: IBM SDK for Java 7 - Using Heapdump.

Supplementary information for IBM WebSphere Real Time for AIX:

Text (classic) Heapdump file format

The text or classic Heapdump is a list of all object instances in the heap, including object type, size, and references between objects.

Header record

The header record is a single record containing a string of version information.

```
// Version: <version string containing SDK level, platform and JVM build level>
```

Example:

```
// Version: J2RE 7.0 IBM J9 2.6 Linux x86-32 build 20101016_024574_lHdRSr
```

Object records

Object records are multiple records, one for each object instance on the heap, providing object address, size, type, and references from the object.

```
<object address, in hexadecimal> [<length in bytes of object instance, in decimal>]
OBJ <object type> <class block reference, in hexadecimal>
<heap reference, in hexadecimal <heap reference, in hexadecimal> ...
```

The object address and heap references are in the heap, but the class block address is outside the heap. All references found in the object instance are listed, including references that are null values. The object type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see “Java VM type signatures” on page 46. Object records can also contain additional class block references, typically in the case of reflection class instances.

Examples:

An object instance, length 28 bytes, of type java/lang/String:

```
0x00436E90 [28] OBJ java/lang/String
```

A class block address of java/lang/String, followed by a reference to a char array instance:

```
0x415319D8 0x00436EB0
```

An object instance, length 44 bytes, of type char array:

```
0x00436EB0 [44] OBJ [C
```

A class block address of char array:

```
0x41530F20
```

An object of type array of java/util/Hashtable Entry inner class:

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

An object of type java/util/Hashtable Entry inner class:

```
0x4158CD80 0x00000000 0x00000000 0x00000000 0x00000000 0x00421660 0x004381C0
0x00438130 0x00438160 0x00421618 0x00421690 0x00000000 0x00000000 0x00000000
0x00438178 0x004381A8 0x004381F0 0x00000000 0x004381D8 0x00000000 0x00438190
0x00000000 0x004216A8 0x00000000 0x00438130 [24] OBJ java/util/Hashtable$Entry
```

A class block address and heap references, including null references:

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

Class records

Class records are multiple records, one for each loaded class, providing class block address, size, type, and references from the class.

```
<class block address, in hexadecimal> [<length in bytes of class block, in decimal>]
CLS <class type>
<class block reference, in hexadecimal> <class block reference, in hexadecimal> ...
<heap reference, in hexadecimal> <heap reference, in hexadecimal>...
```

The class block address and class block references are outside the heap, but the class record can also contain references into the heap, typically for static class data members. All references found in the class block are listed, including those that are null values. The class type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see "Java VM type signatures" on page 46.

Examples:

A class block, length 32 bytes, for class java/lang/Runnable:

```
0x41532E68 [32] CLS java/lang/Runnable
```

References to other class blocks and heap references, including null references:

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

A class block, length 168 bytes, for class java/lang/Math:

```
0x00000000 0x004206A8 0x00420720 0x00420740 0x00420760 0x00420780 0x004207B0
0x00421208 0x00421270 0x00421290 0x004212B0 0x004213C8 0x00421458 0x00421478
0x00000000 0x41589DE0 0x00000000 0x4158B340 0x00000000 0x00000000 0x00000000
0x4158ACE8 0x00000000 0x4152F018 0x00000000 0x00000000 0x00000000
```

Trailer record 1

Trailer record 1 is a single record containing record counts.

```
// Breakdown - Classes: <class record count, in decimal>,
Objects: <object record count, in decimal>,
ObjectArrays: <object array record count, in decimal>,
PrimitiveArrays: <primitive array record count, in decimal>
```

Example:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

Trailer record 2

Trailer record 2 is a single record containing totals.

```
// EOF:  Total 'Objects',Refs(null) :
<total object count, in decimal>,
<total reference count, in decimal>
(,total null reference count, in decimal>)
```

Example:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM type signatures

The Java VM type signatures are abbreviations of the Java types are shown in the following table:

Java VM type signatures	Java type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L *<fully qualified-class>* ;	*<fully qualified-class>*
[*<type>*	*<type>*[ ] (array of *<type>*)
(*<arg-types>*) *<ret-type>*	method

Using system dumps and the dump viewer

The JVM can generate native system dumps, also known as core dumps, under configurable conditions. System dumps are typically large. Most tools used to analyze system dumps are also platform-specific.

The IBM SDK for Java V7 User guide contains useful guidance on using system dumps and the dump viewer, covering:

- Overview of system dumps
- System dump defaults
- Using the dump viewer
  - Using **jextract**
  - Problems to tackle with the dump viewer

– Commands available in **jdmpview**
– Example session
– **jdmpview** commands quick reference

You can find this information here: IBM SDK for Java 7 - Using system dumps and the dump viewer.

Supplementary information for IBM WebSphere Real Time for AIX:

Commands available in jdmpview

jdmpview is an interactive, command-line tool to explore the information from a JVM system dump and perform various analytic functions.

info jitm
Lists AOT and JIT compiled methods and their addresses:
- Method name and signature
- Method start address
- Method end address

For all other command options,see the IBM SDK for Java V7 User guide.

Tracing Java applications and the JVM

JVM trace is a trace facility that is provided in IBM WebSphere Real Time for AIX with minimal affect on performance. In most cases, the trace data is kept in a compact binary format, that can be formatted with the Java formatter that is supplied.

Tracing is enabled by default, together with a small set of trace points going to memory buffers. You can enable tracepoints at run time by using levels, components, group names, or individual tracepoint identifiers.

The IBM SDK for Java V7 User guide contains detailed information on tracing applications, covering:
- What can be traced
- Types of tracepoint
- Default tracing
- Recording trace data
- Controlling the trace
- Tracing Java applications
- Tracing Java methods

When tracing IBM WebSphere Real Time for AIX you must correctly invoke the real-time JVM when including the trace options. For example, when specifying trace options, type:

```
java -Xgcpolicy:metronome -Xtrace:<options>
```

You can find the IBM SDK for Java V7 information here: Tracing Java applications and the JVM.

JIT and AOT problem determination

You can use command-line options to help diagnose JIT and AOT compiler problems and to tune performance.

Although IBM WebSphere Real Time for AIX shares some common components with the IBM SDK for Java V7, the behavior of JIT and AOT is different. This section covers troubleshooting for JIT and AOT issues on IBM WebSphere Real Time for AIX.

Diagnosing a JIT or AOT problem

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT or AOT compiler is faulty and, if so, *where* it is faulty, you can provide valuable help to the Java service team.

About this task

To determine what methods are compiled when the shared class cache is populated, use the **-Xaot:verbose** option on the admincache command-line. For example:

```
admincache -Xrealtime -Xaot:verbose -populate -aot my.jar -cp <My Class Path>
```

This section describes how you can determine if your problem is compiler-related. This section also suggests some possible workarounds and debugging techniques for solving compiler-related problems.

Disabling the JIT or AOT compiler:

If you suspect that a problem is occurring in the JIT or AOT compiler, disable compilation to see if the problem remains. If the problem still occurs, you know that the compiler is not the cause of it.

About this task

The JIT compiler is enabled by default. The AOT compiler is also enabled, but, is not active unless shared classes have been enabled. For efficiency reasons, not all methods in a Java application are compiled. The JVM maintains a call count for each method in the application; every time a method is called and interpreted, the call count for that method is incremented. When the count reaches the compilation threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently. Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT or AOT compiler or it might be elsewhere in the JVM.

The first step in diagnosing the failure is to determine *where* the problem is. To do this, you must first run your Java program in purely interpreted mode (that is, with the JIT and AOT compilers disabled).

Procedure

1. Remove any **-Xjit** and **-Xaot** options (and accompanying parameters) from your command line.
2. Use the **-Xint** command-line option to disable the JIT and AOT compilers. For performance reasons, do not use the **-Xint** option in a production environment.

What to do next

Running the Java program with the compilation disabled leads to one of the following situations:

- The failure remains. The problem is not in the JIT or AOT compiler. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the compiler.
- The failure disappears. The problem is most likely in the JIT or AOT compiler.

  If you are not using shared classes, the JIT compiler is at fault. If you are using shared classes, you must determine which compiler is at fault by running your application with only JIT compilation enabled. Run your application with the **-Xnoaot** option instead of the **-Xint** option. This leads to one of the following situations:

  – The failure remains. The problem is in the JIT compiler. You can also use the **-Xnojit** instead of the **-Xnoaot** option to ensure that only the JIT compiler is at fault.

  – The failure disappears. The problem is in the AOT compiler.

Selectively disabling the JIT compiler:

If your Java program failure points to a problem with the JIT compiler, you can try to narrow down the problem further.

About this task

By default, the JIT compiler optimizes methods at various optimization levels. Different selections of optimizations are applied to different methods, based on their call counts. Methods that are called more frequently are optimized at higher levels. By changing JIT compiler parameters, you can control the optimization level at which methods are optimized. You can determine whether the optimizer is at fault and, if it is, which optimization is problematic.

You specify JIT parameters as a comma-separated list, appended to the **-Xjit** option. The syntax is **-Xjit**:<*param1*>,<*param2*>=<*value*>. For example:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

runs the HelloWorld program, enables verbose output from the JIT, and makes the JIT generate native code without performing any optimizations.

Follow these steps to determine which part of the compiler is causing the failure:

Procedure

1. Set the JIT parameter **count=0** to change the compilation threshold to zero. This parameter causes each Java method to be compiled before it is run. Use **count=0** only when diagnosing problems, because a lot more methods are compiled, including methods that are used infrequently. The extra compilation uses more computing resources and slows down your application. With **count=0**, your application fails immediately when the problem area is reached. In some cases, using **count=1** can reproduce the failure more reliably.
2. Add **disableInlining** to the JIT compiler parameters. **disableInlining** disables the generation of larger and more complex code. If the problem no longer occurs, use **disableInlining** as a workaround while the Java service team analyzes and fixes the compiler problem.

3. Decrease the optimization levels by adding the **optLevel** parameter, and run the program again until the failure no longer occurs, or you reach the "noOpt" level. For a JIT compiler problem, start with "scorching" and work down the list. The optimization levels are, in decreasing order:
   a. scorching
   b. veryHot
   c. hot
   d. warm
   e. cold
   f. noOpt

What to do next

If one of these settings causes your failure to disappear, you have a workaround that you can use. This workaround is temporary while the Java service team analyze and fix the compiler problem. If removing **disableInlining** from the JIT parameter list does not cause the failure to reappear, do so to improve performance. Follow the instructions in "Locating the failing method" to improve the performance of the workaround.

If the failure still occurs at the "noOpt" optimization level, you must disable the JIT compiler as a workaround.

Locating the failing method:

When you have determined the lowest optimization level at which the JIT or AOT compiler must compile methods to trigger the failure, you can find out which part of the Java program, when compiled, causes the failure. You can then instruct the compiler to limit the workaround to a specific method, class, or package, allowing the compiler to compile the rest of the program as usual. For JIT compiler failures, if the failure occurs with **-Xjit:optLevel=noOpt**, you can also instruct the compiler to not compile the method or methods that are causing the failure at all.

Before you begin

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
gpr0=0000000000000006 gpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
gpr4=0000000000000001 gpr5=0000000080056808 gpr6=0000010002BCCA20 gpr7=0000000000000000
......
Compiled_method=java/security/AccessController.toArrayOfProtectionDomains([Ljava/lang/Object;
Ljava/security/AccessControlContext;)[Ljava/security/ProtectionDomain;
```

The important lines are:

vmState=0x00000000
Indicates that the code that failed was not JVM runtime code.

Module= or Module_base_address=
Not in the output (might be blank or zero) because the code was compiled by the JIT, and outside any DLL or library.

Compiled_method=
Indicates the Java method for which the compiled code was produced.

About this task

If your output does not indicate the failing method, follow these steps to identify the failing method:

Procedure

1. Run the Java program with the JIT parameters **verbose** and **vlog=**<*filename*> added to the **-Xjit** or **-Xaot** option. With these parameters, the compiler lists compiled methods in a log file named <*filename*>.<*date*>.<*time*>.<*pid*>, also called a *limit file*. A typical limit file contains lines that correspond to compiled methods, like:

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   Lines that do not start with the plus sign are ignored by the compiler in the following steps and you can remove them from the file. Methods compiled by the AOT compiler start with `+ (AOT cold)`. Methods for which AOT code is loaded from the shared class cache start with `+ (AOT load)`.
2. Run the program again with the JIT or AOT parameter **limitFile=**(<*filename*>,<*m*>,<*n*>), where <*filename*> is the path to the limit file, and <*m*> and <*n*> are line numbers indicating the first and the last methods in the limit file that should be compiled. The compiler compiles only the methods listed on lines <*m*> to <*n*> in the limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled and no AOT code in the shared data cache for those methods will be loaded. If the program no longer fails, one or more of the methods that you have removed in the last iteration must have been the cause of the failure.
3. Optional: If you are diagnosing an AOT problem, run the program a second time with the same options to allow compiled methods to be loaded from the shared data cache. You can also add the **–Xaot:scount=0** option to ensure that AOT-compiled methods stored in the shared data cache will be used when the method is first called. Some AOT compilation failures happen only when AOT-compiled code is loaded from the shared data cache. To help diagnose these problems, use the **–Xaot:scount=0** option to ensure that AOT-compiled methods stored in the shared data cache are used when the method is first called, which might make the problem easier to reproduce. Please note that if you set the **scount** option to 0 it will force AOT code loading and will pause any application thread waiting to execute that method. Thus, this should only be used for diagnostic purposes. More significant pause times can occur with the **–Xaot:scount=0** option.
4. Repeat this process using different values for <*m*> and <*n*>, as many times as necessary, to find the minimum set of methods that must be compiled to trigger the failure. By halving the number of selected lines each time, you can perform a binary search for the failing method. Often, you can reduce the file to a single line.

What to do next

When you have located the failing method, you can disable the JIT or AOT compiler for the failing method only. For example, if the method java/lang/Math.max(II)I causes the program to fail when JIT-compiled with **optLevel=hot**, you can run the program with:

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

to compile only the failing method at an optimization level of “warm”, but compile all other methods as usual.

If a method fails when it is JIT-compiled at “noOpt”, you can exclude it from compilation altogether, using the **exclude**={<*method*>} parameter:

```
-Xjit:exclude={java/lang/Math.max(II)I}
```

If a method causes the program to fail when AOT code is compiled or loaded from the shared data cache, exclude the method from AOT compilation and AOT loading using the **exclude**={<*method*>} parameter:

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

AOT methods are compiled at the “cold” optimization level only. Preventing AOT compilation or AOT loading is the best approach for these methods.

Identifying JIT compilation failures:

For JIT compiler failures, analyze the error output to determine if a failure occurs when the JIT compiler attempts to compile a method.

If the JVM crashes, and you can see that the failure has occurred in the JIT library (`libj9jit26.so`), the JIT compiler might have failed during an attempt to compile a method.

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00050000
Target=2_30_20051215_04381_BHdSMr (Linux 2.4.21-32.0.1.EL)
CPU=ppc64 (4 logical CPUs) (0xebf4e000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000001
Handler1=0000007FE05645B8 Handler2=0000007FE0615C20
R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBCEE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
......
Module=/home/test/sdk/jre/bin/libj9jit26.so
Module_base_address=0000007FE29A6000
......
Method_being_compiled=com/sun/tools/javac/comp/Attr.visitMethodDef(Lcom/sun/tools/javac/tree/
JCTree$JCMethodDecl;)
```

The important lines are:

vmState=0x00050000
: Indicates that the JIT compiler is compiling code. For a list of vmState code numbers, see the Javadump tags table in the IBM SDK for Java V7 User guide, http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.aix.70.doc/diag/tools/javadump_tags_info.html.

Module=/home/test/sdk/jre/bin/libj9jit26.so
: Indicates that the error occurred in `libj9jit26.so`, the JIT compiler module.

Method_being_compiled=
: Indicates the Java method being compiled.

If your output does not indicate the failing method, use the **verbose** option with the following additional settings:

```
-Xjit:verbose={compileStart|compileEnd}
```

These **verbose** settings report when the JIT starts to compile a method, and when it ends. If the JIT fails on a particular method (that is, it starts compiling, but crashes before it can end), use the **exclude** parameter to exclude it from compilation (refer to "Locating the failing method" on page 50). If excluding the method prevents the crash, you have a workaround that you can use while the service team corrects your problem.

Performance of short-running applications

The IBM JIT compiler is tuned for long-running applications typically used on a server. You can use the **-Xquickstart** command-line option to improve the performance of short-running applications, especially for applications in which processing is not concentrated into a few methods.

-Xquickstart causes the JIT compiler to use a lower optimization level by default and to compile fewer methods. Performing fewer compilations more quickly can improve application startup time. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods selected for compilation to be AOT compiled, which improves the startup time of subsequent runs. **-Xquickstart** might degrade performance if it is used with long-running applications that contain methods using a large amount of processing resource. The implementation of **-Xquickstart** is subject to change in future releases.

You can also try improving startup times by adjusting the JIT threshold (using trial and error). See "Selectively disabling the JIT compiler" on page 49 for more information.

JVM behavior during idle periods

You can reduce the CPU cycles consumed by an idle JVM by using the **-XsamplingExpirationTime** option to turn off the JIT sampling thread.

The JIT sampling thread profiles the running Java application to discover commonly used methods. The memory and processor usage of the sampling thread is negligible, and the frequency of profiling is automatically reduced when the JVM is idle.

In some circumstances, you might want no CPU cycles consumed by an idle JVM. To do so, specify the **-XsamplingExpirationTime**<*time*> option. Set <*time*> to the number of seconds for which you want the sampling thread to run. Use this option with care; after it is turned off, you cannot reactivate the sampling thread. Allow the sampling thread to run for long enough to identify important optimizations.

The Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostic files for a problem event.

Gathering the files that are needed by IBM service can reduce the time taken to solve reported problems. The IBM SDK for Java V7 user guide contains detailed information about using the Diagnostics Collector.

You can find this information here: IBM SDK for Java 7 - The Diagnostics Collector.

Garbage Collector diagnostic data

This section describes how to diagnose garbage collection problems.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing garbage collector problems, covering:

- Verbose garbage collection logging
- Tracing garbage collection using **-Xtgc**

You can find this information here: IBM SDK for Java 7 - Garbage Collector diagnostic data.

Supplementary information about the IBM WebSphere Real Time for AIX Metronome Garbage Collector is provided in the following sections.

Troubleshooting the Metronome Garbage Collector

Using the command-line options, you can control the frequency of Metronome garbage collection, out of memory exceptions, and the Metronome behavior on explicit system calls.

Using verbose:gc information:

You can use the **-verbose:gc** option with the **-Xgc:verboseGCCycleTime=N** option to write information to the console about Metronome Garbage Collector activity. Not all XML properties in the **-verbose:gc** output from the standard JVM are created or apply to the output of Metronome Garbage Collector.

Use the **-verbose:gc** option to view the minimum, maximum, and mean free space in the heap. In this way, you can check the level of activity and use of the heap, and then adjust the values if necessary. The **-verbose:gc** option writes Metronome statistics to the console.

The **-Xgc:verboseGCCycleTime=N** option controls the frequency of retrieval of the information. It determines the time in milliseconds that the summaries are dumped. The default value for N is 1000 milliseconds. The cycle time does not mean that the summary is dumped precisely at that time, but when the last garbage collection event that meets this time criterion passes. The collection and display of these statistics can increase Metronome Garbage Collector pause times and, as N gets smaller, the pause times can become large.

A quantum is a single period of Metronome Garbage Collector activity, causing an interruption or pause time for an application.

Example of verbose:gc output

Enter:

```
java -Xgcpolicy:metronome -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

When garbage collection is triggered, a `trigger start` event occurs, followed by any number of `heartbeat` events, then a `trigger end` event when the trigger is satisfied. This example shows a triggered garbage collection cycle as verbose:gc output:

```
<trigger-start id="25" timestamp="2011-07-12T09:32:04.503" />

<cycle-start id="26" type="global" contextid="26" timestamp="2011-07-12T09:32:04.503" intervalms="984.285" />

<gc-op id="27" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.209">
  <quanta quantumCount="321" quantumType="mark" minTimeMs="0.367" meanTimeMs="0.524" maxTimeMs="1.878"
    maxTimestampMs="598704.070" />
  <exclusiveaccess-info minTimeMs="0.006" meanTimeMs="0.062" maxTimeMs="0.147" />
  <free-mem type="heap" minBytes="99143592" meanBytes="114374153" maxBytes="134182032" />
```

```
  <thread-priority maxPriority="11" minPriority="11" />
</gc-op>

<gc-op id="28" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.458">
  <quanta quantumCount="115" quantumType="sweep" minTimeMs="0.430" meanTimeMs="0.471" maxTimeMs="0.511"
    maxTimestampMs="599475.654" />
  <exclusiveaccess-info minTimeMs="0.007" meanTimeMs="0.067" maxTimeMs="0.173" />
  <classunload-info classloadersunloaded=9 classesunloaded=156  />
  <references type="weak" cleared="660" />
  <free-mem type="heap" minBytes="24281568" meanBytes="55456028" maxBytes="87231320" />
  <thread-priority maxPriority="11" minPriority="11" />
</gc-op>

<gc-op id="29" type="syncgc" timems="136.945" contextid="26" timestamp="2011-07-12T09:32:06.046">
  <syncgc-info reason="out of memory" exclusiveaccessTimeMs="0.006" threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="21290752" bytesAfter="171963656" />
</gc-op>

<cycle-end id="30" type="global" contextid="26" timestamp="2011-07-12T09:32:06.046" />

<trigger-end id="31" timestamp="2011-07-12T09:32:06.046" />
```

The following event types can occur:

<trigger-start ...>
The start of a garbage collection cycle, when the amount of used memory became higher than the trigger threshold. The default threshold is 50% of the heap. The `intervalms` attribute is the interval between the previous `trigger end` event (with id-1) and this `trigger start` event.

<trigger-end ...>
A garbage collection cycle successfully lowered the amount of used memory beneath the trigger threshold. If a garbage collection cycle ended, but used memory did not drop beneath the trigger threshold, a new garbage collection cycle is started with the same context ID. For each `trigger start` event, there is a matching `trigger end` event with same context ID. The `intervalms` attribute is the interval between the previous `trigger start` event and the current `trigger end` event. During this time, one or more garbage collection cycles will have completed until used memory has dropped beneath the trigger threshold.

<gc-op id="28" type="heartbeat"...>
A periodic event that gathers information (on memory and time) about all garbage collection quanta for the time covered. A heartbeat event can occur only between a matching pair of `trigger start` and `trigger end` events; that is, while an active garbage collection cycle is in process. The `intervalms` attribute is the interval between the previous heartbeat event (with `id` -1) and this heartbeat event.

<gc-op id="29" type="syncgc"...>
A synchronous (nondeterministic) garbage collection event. See "Synchronous garbage collections" on page 56

The XML tags in this example have the following meanings:

<quanta ...>
A summary of quantum pause times during the heartbeat interval, including the length of the pauses in milliseconds.

<free-mem type="heap" ...>
A summary of the amount of free heap space during the heartbeat interval, sampled at the end of each garbage collection quantum.

<classunload-info classloadersunloaded=9 classesunloaded=156 />
The number of classloaders and classes unloaded during the heartbeat interval.

<references type="weak" cleared="660 />
The number and type of Java reference objects that were cleared during the heartbeat interval.

Note:

- If only one garbage collection quantum occurred in the interval between two heartbeats, the free memory is sampled only at the end of this one quantum. Therefore the minimum, maximum, and mean amounts given in the heartbeat summary are all equal.
- The interval between two heartbeat events might be significantly larger than the cycle time specified if the heap is not full enough to require garbage collection activity. For example, if your program requires garbage collection activity only once every few seconds, you are likely to see a heartbeat only once every few seconds.
- It is possible that the interval might be significantly larger than the cycle time specified because the garbage collection has no work on a heap that is not full enough to warrant garbage collection activity. For example, if your program requires garbage collection activity only once every few seconds, you are likely to see a heartbeat only once every few seconds.

  If an event such as a synchronous garbage collection or a priority change occurs, the details of the event and any pending events, such as heartbeats, are immediately produced as output.
- If the maximum garbage collection quantum for a given period is too large, you might want to reduce the target utilization using the **-Xgc:targetUtilization** option. This action gives the Garbage Collector more time to work. Alternatively, you might want to increase the heap size with the **-Xmx** option. Similarly, if your application can tolerate longer delays than are currently being reported, you can increase the target utilization or decrease the heap size.
- The output can be redirected to a log file instead of the console with the **-Xverbosegclog:<file>** option; for example, **-Xverbosegclog:out** writes the **-verbose:gc** output to the file *out*.
- The priority listed in `thread-priority` is the underlying operating system thread priority, not a Java thread priority.

Synchronous garbage collections

An entry is also written to the **-verbose:gc** log when a synchronous (nondeterministic) garbage collection occurs. This event has three possible causes:

- An explicit `System.gc()` call in the code.
- The JVM runs out of memory then performs a synchronous garbage collection to avoid an OutOfMemoryError condition.
- The JVM shuts down during a continuous garbage collection. The JVM cannot cancel the collection, so it completes the collection synchronously, and then exits.

An example of a `System.gc()` entry is:

```
<gc-op id="9" type="syncgc" timems="12.92" contextid="8" timestamp=
"2011-07-12T09:41:40.808">
  <syncgc-info reason="system GC" totalBytesRequested="260" exclusiveaccessTimeMs="0.009"
  threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="22085440" bytesAfter="136023450" />
  <classunload-info classloadersunloaded="54" classesunloaded="234" />
```

```
  <references type="soft" cleared="21" dynamicThreshold="29" maxThreshold="32" />
  <references type="weak" cleared="523" />
  <finalization enqueued="124" />
</gc-op>
```

An example of a synchronous garbage collection entry as a result of the JVM shutting down is:

```
<gc-op id="24" type="syncgc" timems="6.439" contextid="19" timestamp="2011-07-12T09:43:14.524">
  <syncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009" threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="56182430" bytesAfter="151356238" />
  <classunload-info classloadersunloaded="14" classesunloaded="276" />
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32" />
  <references type="weak" cleared="53" />
  <finalization enqueued="34" />
</gc-op>
```

The XML tags and attributes in this example have the following meanings:

<gc-op id="9" type="syncgc" timems="6.439" ...
: This line indicates that the event type is a synchronous garbage collection. The `timems` attribute is the duration of the synchronous garbage collection in milliseconds.

<syncgc-info reason="..."/>
: The cause of the synchronous garbage collection.

<free-mem-delta.../>
: The free Java heap memory before and after the synchronous garbage collection in bytes.

<finalization .../>
: The number of objects awaiting finalization.

<classunload-info .../>
: The number of classloaders and classes unloaded during the heartbeat interval.

<references type="weak" cleared="53" .../>
: The number and type of Java reference objects that were cleared during the heartbeat interval.

Synchronous garbage collection due to out-of-memory conditions or VM shutdown can happen only when the Garbage Collector is active. It has to be preceded by a `trigger start` event, although not necessarily immediately. Some heartbeat events probably occur between a `trigger start` event and the `synchgc` event. Synchronous garbage collection caused by `System.gc()` can happen at any time.

Tracking all GC quanta

Individual GC quanta can be tracked by enabling the `GlobalGCStart` and `GlobalGCEnd` tracepoints. These tracepoints are produced at the beginning and end of all Metronome Garbage Collector activity including synchronous garbage collections. The output for these tracepoints looks similar to:

```
03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: globalcount=3

03:44:35.284 0x833cd00 j9mm.91 - GlobalGC end: workstackoverflow=0 overflowcount=0
```

Out-of-memory entries

When the heap runs out of free space, an entry is written to the **-verbose:gc** log before the OutOfMemoryError exception is thrown. An example of this output is:

```
<out-of-memory id="71" timestamp="2011-07-12T10:21:50.135" memorySpaceName="Metronome"
  memorySpaceAddress="0806DFDC"/>
```

By default a Javadump is produced as a result of an OutOfMemoryError exception. This dump contains information about the memory used by your program.

```
NULL
1STSEGTOTAL    Total memory:           4066080 (0x003E0B20)
1STSEGINUSE    Total memory in use:    3919440 (0x003BCE50)
1STSEGFREE     Total memory free:       146640 (0x00023CD0)
```

Metronome Garbage Collector behavior in out-of-memory conditions:

By default, the Metronome Garbage Collector triggers an unlimited, nondeterministic garbage collection when the JVM runs out of memory. To prevent nondeterministic behavior, use the **-Xgc:noSynchronousGCOnOOM** option to throw an OutOfMemoryError when the JVM runs out of memory.

The default unlimited collection runs until all possible garbage is collected in a single operation. The pause time required is usually many milliseconds greater than a normal metronome incremental quantum.

Related information:

Using -Xverbose:gc to analyze synchronous garbage collections

Metronome Garbage Collector behavior on explicit System.gc() calls:

If a garbage collection cycle is in progress, the Metronome Garbage Collector completes the cycle in a synchronous way when `System.gc()` is called. If no garbage collection cycle is in progress, a full synchronous cycle is performed when `System.gc()` is called. Use `System.gc()` to clean up the heap in a controlled manner. It is a nondeterministic operation because it performs a complete garbage collection before returning.

Some applications call vendor software that has System.gc() calls where it is not acceptable to create these nondeterministic delays. To disable all System.gc() calls use the **-Xdisableexplicitgc** option.

The verbose garbage collection output for a System.gc() call has a reason of "system garbage collect" and is likely to have a long duration:

```
<gc-op id="9" type="syncgc" timems="6.439" contextid="8" timestamp="2011-07-12T09:41:40.808">
  <syncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009" threadPriority="11"/>
  <free-mem-delta type="heap" bytesBefore="126082300" bytesAfter="156085440"/>
  <classunload-info classloadersunloaded="14" classesunloaded="276"/>
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32"/>
  <references type="weak" cleared="53"/>
  <finalization enqueued="34"/>
</gc-op>
```

Shared classes diagnostic data

Understanding how to diagnose problems that might occur helps you to use shared classes mode.

The IBM SDK for Java V7 User guide contains useful guidance on diagnosing problems with shared classes, covering:

- Deploying shared classes
- Dealing with runtime bytecode modification
- Understanding dynamic updates

- Using the Java Helper API
- Understanding shared classes diagnostic output
- Debugging problems with shared classes

You can find this information here: IBM SDK for Java 7 - Shared classes diagnostic data.

Using the JVMTI

JVMTI is a two-way interface that allows communication between the JVM and a native agent. It replaces the JVMDI and JVMPI interfaces.

JVMTI allows third parties to develop debugging, profiling, and monitoring tools for the JVM. The interface contains mechanisms for the agent to notify the JVM about the kinds of information it requires. The interface also provides a means of receiving the relevant notifications. Several agents can be attached to a JVM at any one time.

The IBM SDK for Java V7 User guide contains detailed information about using JVMTI, including an API reference section on IBM extensions to JVMTI.

You can find this information here: IBM SDK for Java 7 - Using JVMTI.

Using the Diagnostic Tool Framework for Java

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools. DTFJ works with data from a system dump or Javadump.

The IBM SDK for Java V7 User guide contains detailed information about DTFJ. Follow this link: Using the Diagnostic Tool Framework for Java

Chapter 10. Reference

This set of topics lists the options and class libraries that can be used with WebSphere Real Time for AIX

Command-line options

You can specify options on the command line while you are starting Java. Default options have been chosen for best general use.

Specifying Java options and system properties

There are three ways to specify Java properties and system properties.

About this task

You can specify Java options and system properties in these ways. In order of precedence, they are:

1. By specifying the option or property on the command line. For example:
   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```
2. By creating a file that contains the options, and specifying this file on the command line using the **-Xoptionsfile=**<*filename*> option.
   In the options file, specify each option on a new line; you can use the '\' character as a continuation character if you want a single option to span multiple lines. Use the '#' character to define comment lines. You cannot specify **-classpath** in an options file. Here is an example of an options file:
   ```
   #My options file
   -X<option1>
   -X<option2>=\
   <value1>,\
   <value2>
   -D<sysprop1>=<value1>
   ```
3. By creating an environment variable called **IBM_JAVA_OPTIONS** containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

The last option you specify on the command line has precedence over first option. For example, if you specify the options **-Xint -Xjit myClass**, the option **-Xjit** takes precedence over **-Xint**.

System properties

System properties are available to applications, and help provide information about the runtime environment.

com.ibm.jvm.realtime

This property enables Java applications to determine if they are running within a WebSphere Real Time for AIX environment.

If your application is running within the IBM WebSphere Real Time for RT Linux runtime environment, and was started with the **-Xrealtime** option, the **com.ibm.jvm.realtime** property has the value "hard".

© Copyright IBM Corp. 2003, 2014

If your application is running within the IBM WebSphere Real Time for RT Linux runtime environment, but was not started with the **-Xrealtime** option, the **com.ibm.jvm.realtime** property is not set.

If your application is running within the IBM WebSphere Real Time runtime environment, the **com.ibm.jvm.realtime** property has the value "soft".

Standard options

The definitions for the standard options.

-agentlib:<*libname*>**[=**<*options*>**]**
Loads native agent library <*libname*>; for example **-agentlib:hprof**. For more information, specify **-agentlib:jdwp=help** and **-agentlib:hprof=help** on the command line.

-agentpath:*libname***[=**<*options*>**]**
Loads native agent library by full path name.

-assert Prints help on assert-related options.

-cp or -classpath <*directories and .zip or .jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and **CLASSPATH** is not set, the user classpath is, by default, the current directory (.).

-D<property_name>=<*value*>
Sets a system property.

-help or -?
Prints a usage message.

-javaagent:<*jarpath*>**[=**<*options*>**]**
Loads Java programming language agent. For more information, see the java.lang.instrument API documentation.

-jre-restrict-search
Includes user private JREs in the version search.

-no-jre-restrict-search
Excludes user private JREs in the version search.

-showversion
Prints product version and continues.

-verbose:[*class***,***gc***,***dynload***,***sizes***,***stack***,***jni***]**
Enables verbose output.

-verbose:class
Writes an entry to stderr for each class that is loaded.

-verbose:gc
See "Using verbose:gc information" on page 54.

-verbose:dynload
Provides detailed information as each class is loaded by the JVM, including:
- The class name and package
- For class files that were in a `.jar` file, the name and directory path of the `.jar`
- Details of the size of the class and the time taken to load the class

The data is written out to stderr. An example of the output follows:

```
<Loaded java/lang/String from /myjdk/sdk/jre/lib/ppc64/
softrealtime/jclSC160/vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

Note: Classes loaded from the shared class cache do not appear in **-verbose:dynload** output. Use **-verbose:class** for information about these classes.

-verbose:sizes
Writes information to stderr describing the amount of memory used for the stacks and heaps in the JVM

-verbose:stack
Writes information to stderr describing Java and C stack usage.

-verbose:jni
Writes information to stderr describing the JNI services called by the application and JVM.

-version
Prints out version information for the non-real-time mode.

-version:<*value*>
Requires the specified version to run.

-X Prints help on nonstandard options.

Non-standard options

Options that are prefixed by **-X** are nonstandard and subject to change without notice.

The IBM SDK for Java V7 User guide contains detailed information on non-standard options. You can find this information here: IBM SDK for Java 7 - Command-line options.

Supplementary information for IBM WebSphere Real Time for AIX is provided in the following sections.

Metronome Garbage Collector options

The definitions of the Metronome Garbage Collector options.

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM
One occasion when garbage collection occurs is when the heap runs out of memory. If there is no more free space in the heap, using **-Xgc:synchronousGCOnOOM** stops your application while garbage collection removes unused objects. If free space runs out again, consider decreasing the target utilization to allow garbage collection more time to complete. Setting **-Xgc:nosynchronousGCOnOOM** implies that when heap memory is full your application stops and issues an out-of-memory message. The default is **-Xgc:synchronousGCOnOOM**.

-Xnoclassgc
Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is **-Xnoclassgc**.

-Xgc:targetPauseTime=*N*
Sets the garbage collection pause time, where *N* is the time in milliseconds.

When this option is specified, the GC operates with pauses that do not exceed the value specified. If this option is not specified the default pause time is set to 3 milliseconds. For example, running with **-Xgc:targetPauseTime=20** causes the GC to pause for no longer than 20 milliseconds during GC operations.

-Xgc:targetUtilization=*N*
Sets the application utilization to N%; the Garbage Collector attempts to use at most (100-N)% of each time interval. Reasonable values are in the range of 50-80%. Applications with low allocation rates might be able to run at 90%. The default is 70%.

This example shows the maximum size of the heap memory is 30 MB. The garbage collector attempts to use 25% of each time interval because the target utilization for the application is 75%.

```
java -Xgcpolicy:metronome -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*
Specifies the number of GC threads to run. The default is the number of processor cores available to the process. The maximum value you can specify is the number of processors available to the operating system.

-Xgc:verboseGCCycleTime=N
N is the time in milliseconds that the summary information should be dumped.

Note: The cycle time does not mean that the summary information is dumped precisely at that time, but when the last garbage collection event that meets this time criterion passes.

-Xmx<*size*>
Specifies the Java heap size. Unlike other garbage collection strategies, the real-time Metronome GC does not support heap expansion. There is not an initial or maximum heap size option. You can specify only the maximum heap size.

Default settings for the JVM

Default settings apply to the Real Time JVM when no changes are made to the environment that the JVM runs in. Common settings are shown for reference.

Default settings can be changed using environment variables or command-line parameters at JVM startup. The table shows some of the common JVM settings. The last column indicates how you can change the behavior, where the following keys apply:

- **e** - setting controlled by environment variable only
- **c** - setting controlled by command-line parameter only
- **ec** - setting controlled by both environment variable and command-line parameter, with command-line parameter taking precedence.

The information is provided as a quick reference and is not comprehensive.

JVM setting	Default	Setting affected by
Javadumps	Enabled	ec
Javadumps on out of memory	Enabled	ec
Heapdumps	Disabled	ec

JVM setting	Default	Setting affected by
Heapdumps on out of memory	Enabled	ec
Sysdumps	Enabled	ec
Where dump files are produced	Current® directory	ec
Verbose output	Disabled	c
Boot classpath search	Disabled	c
JNI checks	Disabled	c
Remote debugging	Disabled	c
Strict conformancy checks	Disabled	c
Quickstart	Disabled	c
Remote debug info server	Disabled	c
Reduced signalling	Disabled	c
Signal handler chaining	Enabled	c
Classpath	Not set	ec
Class data sharing	Disabled	c
Accessibility support	Enabled	e
JIT compiler	Enabled	ec
AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)	Enabled	c
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e
Default locale	None	e
Time to wait before starting plug-in	N/A	e
Temporary directory	/tmp	e
Plug-in redirection	None	e
IM switching	Disabled	e
IM modifiers	Disabled	e
Thread model	N/A	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	c
Initial stack size for Java Threads 64-bit. Use: **-Xiss<size>**	2 KB	c
Maximum stack size for Java Threads 64-bit. Use: **-Xss<size>**	256 KB	c
Stack size for OS Threads 64-bit. Use **-Xmso<size>**	256 KB	c
Initial heap size. Use **-Xms<size>**	64 MB	c

JVM setting	Default	Setting affected by
Maximum Java heap size. Use **-Xmx<size>**	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	c
Target time interval utilization for an application. The Garbage collector attempts to use the remainder. Use **-Xgc:targetUtilization=<percentage>**	70%	c
The number of garbage collector threads to run. Use **-Xgc:threads=<value>**	The number of processor cores available to the process.	c
Maximum amount of memory that can be allocated to scope memories in **-Xrealtime** mode. Use **-Xgc:scopedMemoryMaximumSize=<size>**.	8 MB	c
Sets the size of the immortal memory area in **-Xrealtime** mode. Use **-Xgc:immortalMemorySize=<size>**	16 MB	c

Note: "available memory" is either the amount of real (physical) memory, or the **RLIMIT_AS** value, whichever is the smallest value.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

© Copyright IBM Corp. 2003, 2014

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

- JIMMAIL@uk.ibm.com [Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Privacy Policy Considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see: (i) IBM’s Privacy Policy at http://www.ibm.com/privacy ; (ii) IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details (in particular, the section entitled “Cookies, Web Beacons and Other Technologies”);

and (iii) the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel and Itanium are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

C

D

E

F

G

H

I

J

L

M

N

O

P

© Copyright IBM Corp. 2003, 2014

R

S

T

U

W

Printed in USA