



BEA JRockit™ SDK

Tuning BEA JRockit JVM

Version 1.4.2
July 2005

Copyright

Copyright © 2004 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Contents

Introduction to Tuning BEA JRockit JVM

How BEA JRockit is Tuned	1-1
JVM Tuning Terminology	1-1
What You'll Find in Tuning BEA JRockit JVM	<i>1-2</i>

Tuning BEA JRockit JVM

Setting the Heap Size	2-2
Setting the Initial and Minimum Heap Size	2-2
Default	2-2
Setting the Maximum Heap Size	2-2
Encountering OutOfMemory Errors	2-2
Default	2-3
Setting the Size of the Nursery	2-3
Default	2-3
Heap Sizing Guidelines	2-3
Defining When a Memory Space will be Cleared	2-4
Default	2-5
Setting the Thread Stack Size	2-5
Minimum Thread Size	2-5
Default	2-5

Basic Tuning Tips and Techniques

Determine What You Want to Tune For	3-1
-----------------------------------------------	-----

Set the Heap Size	3-2
Tune the JVM	3-2
Tuning for High Responsiveness	3-2
Tuning for High Performance	3-3
Other Tuning Tips	3-3
Analyze the Performance by Using the JRA	3-3
Analyze Garbage Collection and Pause Times	3-3
Use -Xgcreport	3-3
Use -Xverbose:memory	3-5
Disable Lock Optimization if Your Application is Slow	3-5

Analyzing and Improving Application Performance

Alternate Methods of Analysis	4-1
Step 1: Find the Hotpaths	4-2
Find the Bottleneck Methods	4-2
Cluster the Bottleneck Methods Together into Hotpaths	4-3
Step 2: Prioritize the Hotpaths	4-3
Step 3: Fix the Hotpath	4-3
Step 4: Repeat Steps 1-3	4-4

Index

Introduction to Tuning BEA JRockit JVM

BEA JRockit JVM automatically adapts to its underlying hardware and to the application running on it. You might wonder, why would anyone need to tune the JVM? The answer is that there are some things BEA JRockit JVM cannot know about your system. For example, how much memory do you want the JVM to use? You probably don't want the JVM to use most of the available memory. Or, how long should the maximum pauses be, to work best within the tolerances of your application? This guide will help answer those questions.

This Introduction includes information on the following subjects:

- [How BEA JRockit is Tuned](#)
- [JVM Tuning Terminology](#)
- [What You'll Find in Tuning BEA JRockit JVM](#)

How BEA JRockit is Tuned

BEA JRockit JVM has a number of non-standard startup parameters, called -x options, that allow you to better tune the JVM for your specific application. This guide documents the different startup parameters and what you need to know about setting them to be able to tune the JVM to ensure optimal performance for your application.

JVM Tuning Terminology

Before continuing, there are some terms you should understand. You may already be familiar with some of the terms, especially if you have read any other documents about garbage collectors.

Garbage collector

The garbage collector is the key to effectively managing BEA JRockit's memory system, which is the ultimate goal of JVM tuning. Garbage collection is the process of clearing dead objects from the heap, thus releasing that space for new objects.

Memory throughput

Memory throughput measures the time between when an object is no longer referenced and the time that it's reclaimed and returned as free memory. The higher the memory throughput the shorter is the time between the two events. Moreover, the higher the memory throughput the smaller the heap you will need.

Pause time

Garbage collector pause time is the length of time that the garbage collector stops all Java threads during a garbage collection. The longer the pause, the more unresponsive your system will be. The worst pause time and the average pause time are the two most interesting values you can use for tuning the system.

Thread-local allocation

Thread-local allocation removes object allocation contention and reduces the need to synchronize between thread performing allocations on the heap. It also gives increased cache performance on a multi-CPU system, because it reduces the risk of two threads running on different CPUs having to access the same memory pages at the same time.

Thread-local allocation is not the same thing as thread-local objects, but many people tend to confuse the two terms. Thread-local allocation does not determine whether the objects can be accessed from a single thread only (that is, thread-local objects); thread-local allocation means that the thread has an area of its own where no other thread will create new objects. The objects that the thread creates in that area may still be reached from other threads.

What You'll Find in Tuning BEA JRockit JVM

This guide is divided into three sections:

- [Tuning BEA JRockit JVM](#) describes the basic tuning parameters for the JVM. The instructions in this section describe default and optimal heap and nursery settings and how to use them to tune the JVM.
- [Basic Tuning Tips and Techniques](#) contains some helpful hints for maximizing system performance by tuning BEA JRockit to provide either optimal memory throughput or minimal garbage collection pause times.

- [Analyzing and Improving Application Performance](#) shows you how to you can improve application performance by uncovering “hotpaths,” or bottlenecks in processing, and either working around them or eliminating them completely.

Tuning BEA JRockit JVM

Have you ever seen strange pauses in your application that you haven't been able to explain? Have you seen one or all CPUs pegged on 100% utilization and all the others on 0% and still very few transactions in your system? If you answered yes to either of these two questions, your application might have been suffering from the effects of a poorly performing garbage collector. Some fairly simple tuning of the memory management system can improve performance dramatically for many applications.

To provide the optimal out-of-the-box experience, BEA JRockit JVM comes with default values that adapt automatically to the specific platform on which you are running BEA JRockit JVM. Tuning BEA JRockit JVM is accomplished by using non-standard—or -x—command line options that you enter at startup. -x options are exclusive to BEA JRockit JVM. Use them to set the behavior of BEA JRockit JVM to better suit the needs of your Java applications.

This section describes how to use these options to tune BEA JRockit. It includes information on the following subjects:

- [Setting the Heap Size](#)
- [Defining When a Memory Space will be Cleared](#)
- [Setting the Thread Stack Size](#)

Note: If BEA JRockit behaves in some unexpected way, please consult the [BEA JRockit Developers FAQ](#). If that doesn't solve your problem, please send an e-mail to support@bea.com

Setting the Heap Size

System performance is greatly influenced by the size of the Java heap available to the JVM. This section describes the command line options you use to define the initial and maximum heap sizes and the size of any nursery that might be required by generational garbage collectors. It also includes key guidelines for help you determine the optimal heap size for your BEA JRockit implementation.

Setting the Initial and Minimum Heap Size

`-Xms<size>`

`-Xms` sets the initial and minimum size of the heap. For this, we recommend that you set it to the same size as the maximum heap size; for example:

```
-java -Xgcprio:throughput -Xmx:64m -Xms:64m myClass
```

Default

`-server` mode: 25% of the amount of free physical memory in the system, up to 64 MB and a minimum of 8 MB.

`-client` mode: 25% of the amount of free physical memory in the system, up to 16 MB and a minimum of 8 MB.

Setting the Maximum Heap Size

`-Xmx:<size>`

`-Xmx` sets the maximum size of the heap. Use the following guidelines to determine this value:

- On IA32 the maximum possible heap size is about 1.8 GB (which is the largest contiguous address space the O/S will give a process).
- Because IA64 machines have a larger address space, the 1.8 GB limit does not apply.
- Typically, for any platform you don't want to use a larger maximum heap size setting than 75% of the *available* physical memory. This is because you need to leave some memory space available for internal usage in the JVM.

Encountering OutOfMemory Errors

If you encounter OutOfMemory errors, you should increase the maximum heap size according to the guidelines listed above.

Default

`-server` and `-client` modes: The default value is the lesser of 75% of the total physical memory up to 1536 MB.

Setting the Size of the Nursery

`-Xns:<size>`

`-Xns` sets the size of the young generation (nursery) in generational garbage collectors.

Optimally, you should try to make the nursery as large as possible while still keeping the garbage collection-pause times acceptably low. This is particularly important if you are creating a lot of temporary objects.

Note: To display pause times, include the option `-Xgcpause` when you start BEA JRockit JVM. The maximum size of a nursery *cannot* exceed 95% of the maximum heap size.

Default

`-server` mode: the default nursery size is 10 MB per CPU; for example, the default for a ten CPU system would be 100 MB.

`-client` mode: the default nursery size is 2 MB.

Additionally, the default nursery will never exceed 25% of maximum heap size, unless you use `-Xns` to explicitly set it to something larger.

Heap Sizing Guidelines

The following guidelines offer some hints on how best to size a heap to achieve optimal performance. Be aware that these guidelines are mainly valid for the `-server` (default) start-up option (see [“Setting the Default Garbage Collector”](#)).

- To get a fixed heap size—for example, if you want a controlled environment—set `-Xms` and `-Xmx` to the same value.
- To improve start-up performance, set `-Xms` to *at least* the approximate amount of live data. You can set `-Xms` to as much as twice the minimum amount of live data without disturbing automatic heap resizing. If `-Xmx` isn't set, or is set too low, frequent garbage collections can slow startup until BEA JRockit has grown the heap.

- To avoid paging, do not set `-Xmx` higher than the amount of available physical memory in the system. Also, you must account for the memory usage of other applications intended to run simultaneously with the JVM, as these will impact memory availability.
- If the amount of free memory in the system varies widely, you might not want to set `-Xmx` at all. This will prevent BEA JRockit from growing the heap when there is too little memory in the system. Be aware that this will throw an `OutOfMemoryError` if object allocation fails with the current heap size and the heap cannot grow without causing paging.
- Paging might occur even if `-Xmx` isn't set. BEA JRockit will not shrink the heap if more than half the heap is filled with live data. Thus, BEA JRockit might not always be able to shrink the heap if the amount of free memory is reduced after JRockit has been started; for example, when another application is started.
- Setting a low maximum heap (`-Xmx`) compared to the amount of live data can affect performance by forcing BEA JRockit to perform frequent garbage collections. If you anticipate a “tight” heap (that is, the amount of live objects is close to the size of the heap) and the application allocates many short lived objects, we suggest you use a generational garbage collector (`-Xgc:gencon`) instead of a single-spaced garbage collector (`-Xgc:singlecon` and `-Xgc:parallel`).

Defining When a Memory Space will be Cleared

`-Xcleartype:<gc|local|alloc>`

`-Xcleartype` defines when the memory space occupied by an object that has been garbage collected will be cleared. When clearing is actually performed is specified by the selected parameter, as described in [Table 2-1](#).

Table 2-1 `-Xcleartype` Parameters

Use this parameter...	To clear space...
<code>gc</code>	During the garbage collection
<code>local</code>	When a thread-local area is allocated
<code>alloc</code>	When that space is allocated for a new object
This is the preferred option if the objects allocated are very large (1 to 2 kilobytes). The <code>alloc</code> parameter is currently not available on IA64 systems.	

The preferable options are either `alloc` or `local`.

Default

If the clear type is not set, the default is `alloc` on IA32 systems and `gc` on IA64 systems.

Setting the Thread Stack Size

`-Xss<size>[k|K] [m|M]`

`-Xss<size>[k|K] [m|M]` sets the thread stack size in kilobytes.

Minimum Thread Size

Minimum thread stack size is 16 kilobytes. If `-xss` is set below the minimum value, thread stack size will default to the minimum value automatically.

Default

If the thread stack size has not been set the default value depends on the platform on which BEA JRockit is running. [Table 2-2](#) shows these defaults:

Table 2-2 Default Head Stack Sizes

O/S	32-bit Default	64-bit Default
Windows	64 kB	320 kB
Linux	128 kB	1 mB

Basic Tuning Tips and Techniques

When you install BEA JRockit JVM, it includes a host of default start-up options that ensure a satisfactory out-of-the-box experience; however, often, these options might not provide your application with the optimal performance you should experience with BEA JRockit JVM. Therefore, BEA JRockit JVM comes with numerous alternative options and algorithms to suit different applications. This section describes some of these options and some basic tuning techniques you can use at startup. It includes information on the following subjects:

- [Determine What You Want to Tune For](#)
- [Set the Heap Size](#)
- [Tune the JVM](#)
- [Other Tuning Tips](#)

Note: The tuning settings discussed in this section refer to standard and non-standard tuning options which are not thoroughly described in the present context. For more information on these options, please refer to [Tuning BEA JRockit JVM](#).

Determine What You Want to Tune For

Before you start BEA JRockit JVM, you need to determine these two factors:

- How much of your machine memory do you want BEA JRockit JVM to use?
- What do you want from BEA JRockit JVM, the highest possible responsiveness or the highest possible performance?

Once you've answered these questions, use the information provided below to tune BEA JRockit JVM to achieve those goals.

Set the Heap Size

Generally, you want to set the maximum heap size as high as possible, but not so high that it causes page-faults for the application or for some other application on the same computer. Heap sizing is accomplished by using the `-Xms` (minimum heap size) and `-Xmx` (maximum heap size) options. For details on these options and guidelines for sizing the heap, please refer to “[Setting the Heap Size](#)” in [Tuning BEA JRockit JVM](#).

Tune the JVM

As mentioned above, you need to consider how you want BEA JRockit to perform: for the highest possible responsiveness or the highest possible performance? This section describes how to tune for either type of performance.

Tuning for High Responsiveness

If you want the highest responsiveness from your application and guarantee minimal pause times, do the following:

- Select a garbage collector that suits your application best:
 - Use the unified garbage collector and set `-Xgcprio:pausetime`.
- OR
- If you want to use a fixed garbage collector, select the [Generational Concurrent](#) garbage collector (`-Xgc:gencon`).
- Set the initial (`-Xms`) and maximum (`-Xmx`) heap sizes, as described in [Set the Heap Size](#). If you're using a fixed, generational concurrent garbage collector, a larger heap reduces the frequency of garbage collection and will allow collection at less intrusive points. This will prevent longer pauses.
- Set the size of the nursery (`-Xns`).

If you are creating a lot of temporary objects you should have a large nursery. Larger nurseries usually result in slightly longer pauses, so, while you should try to make the nursery as large as possible, don't make it so large that pause times are unacceptable. You can see the nursery pause times in BEA JRockit JVM by starting the JVM with `-Xgcpause`.

Tuning for High Performance

If you want the highest possible performance BEA JRockit can provide, you will want to optimize memory throughput. Set these tuning options at startup:

- Select a garbage collector that suits your application best:
 - Select the unified garbage collector with the throughput priority specified (`-Xgcprio:throughput`)

OR

- Select the [Parallel](#) garbage collector. A parallel garbage collector doesn't use a nursery, so you don't need to set `-Xns`.
- Set the largest initial (`-Xms`) and maximum (`-Xmx`) heap sizes that your machine can tolerate, as described in [Set the Heap Size](#).

Other Tuning Tips

This section describes other practices you can employ to improve BEA JRockit JVM performance.

Analyze the Performance by Using the JRA

The JRockit Runtime Analyzer (JRA) is a great way to look at the performance of JRockit. The JRA records what happens in your system in runtime and then saves the findings in a file that can be analyzed through a separate JRA tool. The recording contains information about, for example, memory usage, Java heap content, and hot methods. For information on how to use the JRA, see:

[Using the JRockit Runtime Analyzer](#)

Analyze Garbage Collection and Pause Times

Analyzing garbage collection and pause times together will give you a good idea of how well your application is performing while running with BEA JRockit JVM.

Use `-Xgcreport`

Use the option `-Xgcreport` to generate an end-of-run report that shows the garbage collection statistics. You can use this report to determine if you're using the most effective garbage collector for your application. As shown in [Listing 3-1](#), the `-Xgcreport` shows a detailed profile of

collections on both the nursery and the old generation (in this case, the garbage collector was generational).

Listing 3-1 -Xgcreport Output: Generational Garbage Collector

```
[memory ]
[memory ] Memory usage report
[memory ]
[memory ] young collections
[memory ] number of collections = 201
[memory ] total promoted =      395672 (size 11807976)
[memory ] max promoted =      3797 (size 113720)
[memory ] total GC time =      5.994 s
[memory ] mean GC time =      29.819 ms
[memory ] maximum GC Pauses =    48.175 , 54.541, 81.423 ms
[memory ]
[memory ] old collections
[memory ] number of collections = 24
[memory ] total promoted =      0 (size 0)
[memory ] max promoted =      0 (size 0)
[memory ] total GC time =      4.083 s (pause 1.812 s)
[memory ] mean GC time =      170.125 ms (pause 75.498 ms)
[memory ] maximum GC Pauses =    0.489 , 2.213, 99.671 ms
[memory ]
[memory ] number of concurrent mark phases = 7
[memory ] number of parallel mark phases = 17
[memory ] number of concurrent sweep phases = 8
[memory ] number of parallel sweep phases = 16
```

By using this report, you can determine where performance is being impacted during garbage collection. For example, you might determine that pause times are too long change from a static garbage collector to a dynamic one that attempts to minimize pause time by setting `-Xgcprio:pausetime`.

For more information on using `-Xgcreport`, see [“Viewing Garbage Collection Activity” in Using the BEA JRockit Memory Management System](#)

Use `-Xverbose:memory`

Use the option `-Xverbose:memory` to display the pause times for every garbage collection during a run. Note that this option is used mainly for debugging purposes as it creates a lot of output to the console. For information on using `-Xverbose`, please refer to “[Displaying Logging Information](#)” in [Starting and Configuring BEA JRockit JVM](#).

Disable Lock Optimization if Your Application is Slow

If your application consists of a fairly small active portion (less than a couple of 100 lines of code being accessed more than 80% of the time) and the application is heavily multithreaded with multiple threads accessing the active portion, you may be able to speed up your application by specifying `-XXdisablefatspin`. This option disables a lock optimization in JRockit.

Basic Tuning Tips and Techniques

Analyzing and Improving Application Performance

This section describes how you can improve application performance by uncovering “hotpaths,” or bottlenecks in processing, and either working around those hotpaths or eliminating them completely.

Analyzing and improving your application is a four-step process:

- [Step 1: Find the Hotpaths](#)
- [Step 2: Prioritize the Hotpaths](#)
- [Step 3: Fix the Hotpath](#)
- [Step 4: Repeat Steps 1-3](#)

Alternate Methods of Analysis

The method presented here differs somewhat from the normal use of Java profilers you might have encountered. The reason for the difference is to provide a more accurate picture of what the application is doing. All kinds of profiling are intrusive, and thus change the behaviour of the application you are observing. This causes you to look at data that is not really representative of the application's behaviour.

One way to minimize the impact of profiling is to collect less data (use sampling instead of callgraph analysis), but then this data may not tell you enough to find problems. Therefore, you should can then combine the exact data with information taken during a more intrusive run. This will give you a better picture of what the application is doing. To do this, follow these steps:

- First collect sampling data to find out where in the application we are spending too much time.
- Next, combine this information with callgraph data to find out how we came to that location.

Note: While this method provides a good way to find out accurate profiles of your application, a more thorough analysis might still be needed.

Step 1: Find the Hotpaths

Finding Hotpaths is a two-step process:

- [Find the Bottleneck Methods](#)
- [Cluster the Bottleneck Methods Together into Hotpaths](#)

Find the Bottleneck Methods

As their name implies, bottleneck methods are those methods that require excessive time and processing resources to execute. These bottlenecks can greatly affect system performance and need to be identified. To find bottleneck methods, do the following:

- Use the Intel VTune profiling tool with JRockit to analyze performance. VTune uses features on the processor to gather information about which code the processor is currently executing. This is done after a fixed number of either clock ticks (wall clock time) or after a fixed number of instructions retired (actual instructions executed in the processor). VTune then uses this data together with symbol information from Java code to present information about where the application spends most of its time.

Note: You should use VTune only in the “sampling” mode, not the “call-graph” mode.

For more information on how to use VTune, please refer to the appropriate vendor documentation at:

<http://www.intel.com/software/products/vtune/>

- Use the Java Virtual Machine Profiling Interface (JVMPI). JVMPI is a two-way function call interface between the Java virtual machine and an in-process profiler agent. On one hand, the VM notifies the profiler agent of various events, corresponding to, for example, heap allocation, thread start, and so on. Concurrently, the profiler agent issues controls and requests for more information through the JVMPI.
 - From inside VTune, start this interface by setting the `-Xrunjavaperf` option:

`-Xrunjavaperf`

- To reduce the profiler overhead, use the `-Xjvmpi` option:

`-Xjvmpi:allocs=off,monitors=off,entryexit=off.`

For a list of recommended `-Xjvmpi` settings, please refer to [Table 4.1](#) in [Profiling and Debugging with BEA JRockit](#)

Cluster the Bottleneck Methods Together into Hotpaths

To cluster the bottleneck methods together into hotpaths, do the following:

1. Run your favorite profiler, such as OptimizeIt or JProbe.
2. Review the call-traces produced by it.
3. Combine these call-traces with the bottleneck methods discovered in [Find the Bottleneck Methods](#), above. This combination of bottleneck methods and call-traces will identify your hotpaths.

Step 2: Prioritize the Hotpaths

To prioritize the hotpaths, do the following:

1. Sum all of the time spent in each hotpath to compute a total hotpath time.
2. Remove all individual hotpaths that represent less than a prescribed percentage of the total hotpath time; for example, a good initial percentage might be 5%. This is the hotpath threshold.
3. Ignore any hotpath that falls below the hotpath threshold. Any hotpath time above the threshold should be optimized.

Step 3: Fix the Hotpath

You need to rely on your own judgement and knowledge of the application to fix hotpaths. Once you've identified and prioritized the hotpaths, look at each one and decide if the code is really needed or if you can make some simple changes, perhaps to the coding or to an algorithm, to avoid it or eliminate it as a hotpath. If you determine that you cannot remove the hotpath, what can you do to make it faster? Rewrite the code so it's more efficient?

Also, are you sure that anything you do will actually improve performance. Any optimization you attempt should at least double performance of the hotpath or your efforts might be wasted. For

example, a performance increase of 5% or a hotpath that takes only 5% of the time is only going to improve performance .25%.

Step 4: Repeat Steps 1-3

Continue repeating the optimization process until you attain the desired system performance.

Index

C

- command line options
 - Xcscartype 2-4
 - Xgcpcase 2-3, 3-2
 - Xms 2-2
 - Xmx 2-2
 - Xns 2-3
 - Xss 2-5

D

- default values, thread system 2-1

G

- garbage collection
 - concurrent
 - generational concurrent 2-3
 - young generation 2-3
- garbage collector 2-1

H

- heap
 - size 2-2
- Hotpath 4-2
- hotpath 1-3, 4-3
 - threshold 4-3
- total hotpath time 4-3

I

- IA32 2-5
- IA64 2-5

- IA64, limitations 2-4

J

- JRockit Runtime Analyzer 3-3
- JVMPI 4-2, 4-3

N

- nursery 2-3

P

- profiler agent 4-2

S

- starting JRockit 2-1
- Support 2-1

T

- thread system
 - default values 2-1

