BEA WebLogic JRockit™ 7.0 SDK

Performance Tuning Guide

Release 7.0 Service Pack 5
March 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

CHAPTER

1 Introduction

BEA WebLogic JRockit 7.0 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 WebLogic JRockit 7.0 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?

WebLogic JRockit 7.0 JVM has a number of non-standard startup parameters, called `-X` options, that allow you to better tune the JVM for your specific application. In WebLogic JRockit 7.0 JVM there are two main subsystems that can be optimized separately using different startup parameters: the memory management system (including the garbage collectors), and the thread system. This guide documents the different startup parameters and what you need to know about these subsystems to be able to tune them efficiently. You will find that the memory management system is the subsystem that gives you the most tuning opportunities. By tuning these parameters you will likely find the best performance improvements for your application.

2 Tuning the WebLogic JRockit 7.0 JVM Memory Management System

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.

This section includes information on the following subjects:

Memory Management 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.

- **Generational garbage collector**

  A generational garbage collector divides the memory into two or more areas called “generations”. Instead of allocating objects in one single space and garbage collecting that whole space when it gets full, most of the objects are allocated in the “young generation”, called the nursery. As most objects die young, most of the time it will be sufficient to garbage collect only the nursery and not the entire heap.

- **Concurrent garbage collector**

  A concurrent garbage collector does its work in parallel with ordinary work; that is, it does not stop all Java threads to do the complete garbage collection. Most garbage collectors today are “stop-the-world” or parallel collectors; these are not very efficient. Using a parallel collector, if you have to garbage collect the whole of a large heap there could be a pretty long pause, up to several seconds, depending on the heap size.

- **Parallel garbage collector**

  A parallel garage collector is a garbage collector that stops all java threads completely (stop-the-world) during the whole garbage collection and uses all available CPUs to perform the collection.

- **Thread-local allocation**

  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 (i.e., 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. 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.

- **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.

- **Memory throughput**

  Memory throughput measures the time it takes between when an object is no longer referenced and the time it gets 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.

WebLogic JRockit 7.0 JVM Garbage Collectors

This section describes the four garbage collectors available in WebLogic JRockit 7.0 JVM.

Generational Copying

The first type of WebLogic JRockit 7.0 JVM garbage collector is the generational copying garbage collector (`-Xgc:gencopy`). It is specifically designed as a lightweight alternative for use on single CPU systems with a small (less then 128 MB) heap. It is suitable for testing applications on your desktop machine; however for a deployment environment another garbage collector would in most cases be more efficient.

Single Spaced Concurrent

The second type of WebLogic JRockit 7.0 JVM garbage collector is the single spaced concurrent garbage collector (`-Xgc:singlecon`). What is unique about the concurrent garbage collectors is that they remove garbage collection pauses completely. Using these garbage collectors, the heaps can be gigabyte-size and there will be no long pauses. However, keep in mind that concurrent garbage collectors trade memory throughput for reduced pause time. It takes longer between the time the object is referenced the last time and the system detects and reclaims it; in other words it takes longer for the object to die. The natural consequence of this is that you will most likely need a larger heap with a concurrent garbage collector than you need with any other. In addition, if your ordinary Java threads create more garbage than the concurrent garbage collector manages to collect, there will be pauses while the Java threads are waiting for the concurrent garbage collector to complete its cycle.

Generational Concurrent

The third type of WebLogic JRockit 7.0 JVM garbage collector is the generational concurrent garbage collector (`-Xgc:gencon`). In this garbage collector, objects are allocated in the young generation. When the young generation (called a nursery) is full, WebLogic JRockit 7.0 JVM "stops-the-world" and moves the objects that are still live in the young generation to the old generation. An old collector thread runs in the background all the time; it marks objects in the old space as live and removes the dead objects, returning them to the JVM as free space.

The advantage of the generational concurrent garbage collector compared to the single spaced concurrent garbage collector is that it has a higher memory throughput.

Parallel

The fourth type of WebLogic JRockit JVM garbage collector is the parallel garbage collector (`-Xgc:parallel`). When the heap is full, all Java threads are stopped and every CPU is used to perform a complete garbage collection of the entire heap. A parallel collector can have longer pause times than concurrent collectors, but it maximizes throughput. Even on single CPU machines, this maximized performance

makes parallel the recommended garbage collector, provided that your application can tolerate the longer pause times.

Monitoring Garbage Collection

The option `-Xgcreport` causes WebLogic JRockit 7.0 JVM to print a comprehensive garbage collection report at program completion. The option `-Xgcpause` causes WebLogic JRockit 7.0 JVM to print a line each time Java threads are stopped for garbage collection. Combining the two is a very good way of examining the memory behavior of your application.

More Memory Management Options

The following options allow you to manage your memory more efficiently.

- `-Xmx:<size>/-Xms<size>`

  `-Xmx` sets the maximum size of the heap. The general recommendation is to set this 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. Set it to something less than the amount of memory in the machine. If you have multiple applications running on the computer at the same time the value could be much lower. The general recommendation is to set the initial heap size (`-Xms`) to the same size as the maximum heap size.

- `-Xns:<size>`

  `-Xns` sets the size of the young generation (nursery). If you are creating a lot of temporary objects you should have a large nursery. Generally, the larger you can make the nursery while keeping the GC-pause times acceptably low, the better. You can see the nursery pause times in WebLogic JRockit 7.0 JVM by starting the JVM with `-Xgcpause`, but you have to decide yourself what is an acceptable GC pause time before your system becomes unresponsive.

- `-Xallocationtype:<global|local>`

`-Xallocationtype` sets the type of thread allocation. The allocation type `local` is recommended for the vast majority of applications. However, if the maximum heap size is very small (less then 128 MB) or if the number of threads used by the application is very high (several hundred) the allocation type `global` might work better, particularly on single CPU systems. The reason for this is that every thread-local area consumes a fixed amount of memory (approximately 2 kilobytes). If the number of threads is very high or the heap size very small when using thread-local allocation the potential waste of space could cause excess fragmentation of the heap. This leads to more frequent garbage collections and may cause the application to run out of memory prematurely.

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

  `-Xcleartype` defines when the memory space occupied by an object that has been garbage collected will be cleared. It can be done during the garbage collection (`gc`), when a thread-local area is allocated (`local`) or when that space is allocated for a new object (`alloc`). It is recommended that you use `local` or `alloc. alloc` may work better if the objects allocated are predominately very large (1 to 2 kilobytes).

Notes:

The `-Xcleartype:local` option is available only if the `-Xallocationtype` is set to local.

On IA64 systems the option `alloc` is not available.

Memory Management System Defaults

This section describes the default values for the WebLogic JRockit 7.0 JVM Memory Management system. To provide the best out-of-the-box performance possible they adapt automatically to the specific platform on which the VM is running.

Heap Size

If the initial heap size (`-Xms`) is not set the initial heap size will be 75% of the free memory. Generally, the default maximum heap size (`-Xmx`) is 75% of the physical memory in the machine. However, when running WebLogic JRockit with a small initial heap (that is, less than about 32MB) the default maximum heap size will depend on the initial heap size. The default maximum heap size is `-Xms`^{2} (in megabytes), *up to* 75% of the physical memory; for example, if `-Xms` is 8MB, the default maximum heap size will be 8MB^2, or 64MB; if `-Xms` is 128MB, the default maximum heap size would be 128^2, or 16384MB (16 GB). Be aware that, if the machine has less physical memory than the value of `-Xms`^{2}, the default maximum heap will be restricted to 75% of the *physical memory*.

Note: These figures are subject to any platform limitations that determine how much contiguous memory a process can allocate.

Garbage Collector

If the garbage collector (`-Xgc`) has not been set and the maximum heap size (set by using `-Xmx` or using the default as described above) is less than 128 MB, the default garbage collector will be the generational copying (`gencopy`) garbage collector, otherwise the default is the generational concurrent (`gencon`) garbage collector.

Nursery Size

If the nursery size (`-Xns`) has not been set the default size depends on the number of CPUs. For the generational copying (`gencopy`) garbage collector the default nursery size is 320 KB times the number of CPUs and for the generational concurrent (`gencon`) garbage collector the default nursery size is 10 MB times the number of CPUs.

Thread Stack Size

If the thread stack size (`-Xss`) has not been set the default value depends on the threading system and the platform you are running on. When using thin threads the minimum thread stack size is 8 kilobytes and the default is 64 kilobytes. When using native threads the minimum thread stack size is 16 kilobytes. For Windows the default thread stack size when using native threads is 64 kilobytes and for Linux it is 128 kilobytes.

Note: If `-Xss` is set to less than the minimum value, the minimum value will be used automatically.

Allocation Type

If the allocation type (`-Xallocationtype`) is not set, the default is `global` for the generational copying (`gencopy`) garbage collector and `local` for all others (`singlecon`, `gencon`, and `parallel`).

Clear Type

If the clear type (`-Xcleartype`) is not set the default is `alloc` on IA32 systems and `gc` on IA64 systems.

Note: On IA64 systems the option `alloc` is not available.

3 Tuning the WebLogic JRockit 7.0 JVM Thread System

WebLogic JRockit 7.0 JVM has two different thread systems, native threads and thin threads. The first thing to do when tuning the thread system is to select the thread system that works best for your application. What then are the pros and cons of the two thread systems and why should you prefer one thread system over another? Begin by understanding the differences between the two thread systems. The native thread model is the common threading model that most JVMs use, where each Java thread is mapped to an operating system thread of its own. The thin thread model is a hybrid threading model where WebLogic JRockit 7.0 JVM has a small fixed number of operating system threads and consequently runs multiple Java threads on top of the same operating system thread. Both models are preemptive threading models, so if one thread uses its whole time slice it gets preempted and another Java thread gets to run instead.

What are the advantages of the native thread system? The foremost advantage is that it is standard, so if you have an application that has native code and that native code relies upon the fact that each Java thread is mapped on to a operating system thread of its own, this is the only model that works (both DB2 and Oracle level 2 JDBC database drivers have been known to rely upon this). The second advantage is that on a multiprocessor system when the application has few active threads, the operating system scheduling system is better at utilizing the CPUs efficiently. A disadvantage of using native threads is that context switching is more costly as it has to be done in the operating system instead of only in the JVM. Another disadvantage is that every Java thread consumes more resources, because it requires an operating system thread of its own.

When should you use thin threads? The major benefit of using thin threads is that switching between java threads is a lot cheaper as it can be done inside the VM rather then in the operating system. Therefore the general recommendation is that if there are more than a couple of hundred threads you should try the thin thread model and determine whether it works better for your application. On Linux you should try the thin thread model, especially on a single-CPU system. This is because Linux threads in themselves are very expensive to use. In addition, if the number of threads is high, you should consider `-Xallocationtype:global` as suggested above to reduce heap fragmentation.