

Java Tuning for Performance and IBM POWER6 Support

Venkata Ravi Kumar Dadi
Levon Stepanian

Contributors:

Carlos Cavanna, Brian Hall, Kenneth Ma, Marius Pirvu,
Vijay Sundaresan, Enyu Wang and Julian Wang

IBM Corporation
June 6, 2007





Table of Contents

Abstract	1
Introduction	1
Prerequisites	1
Memory-Related Performance and Tuning.....	2
Tuning the Java Heap for Performance on AIX.....	2
Garbage Collection	3
GC Configurations.....	3
Heap Fragmentation	5
Large Pages.....	5
Thread-related Tuning.....	8
Threading Model.....	8
AIX Threading Models	8
Threads Created by Java.....	8
Scheduling Scope.....	9
Scheduling Policy for Threads.....	9
Time Slice.....	10
Spin Loop Time.....	10
Maxspin.....	11
Yield Loop Time.....	11
AIX Performance Tools.....	11
Exploiting Dynamic Logical Partitions with Java	11
Startup Performance	12
Reducing Java Application Startup Time.....	13
Exploiting Java Shared Classes and Ahead-Of-Time (AOT) Compilation	14
Java Coding Guidelines.....	16
Object Allocation (don't think GC can do everything for you)	16
Classes, Methods and Fields.....	17
Loops and Controls.....	18
Threads and Locks.....	18
POWER6 Hardware Feature Exploitation.....	19
Simultaneous Multithreading.....	19
Decimal Floating-Point Unit Exploitation	20
Java 6 BigDecimal	20
64-bit Decimal Floating-Point in Java 6	21



64-bit Decimal Floating-Point Coding Suggested Practices	21
Preliminary Performance Results	21
Summary	23
Resources	23
Memory-Related Performance and Tuning	23
Tuning the Java Heap	23
Garbage Collection	23
Large Pages	24
Thread-Related Tuning	24
Threading Models	24
Exploiting Dynamic Logical Partitions (Dynamic LPAR) with Java.....	24
Startup Performance	24
Java Coding Guidelines	25
POWER6 Hardware Feature Exploitation	25
Simultaneous Multithreading	25
Decimal Floating-Point Unit Exploitation.....	25
About the Authors	26

Abstract

This paper describes suggested practices for and performance results of using an IBM Software Development Kit (SDK) for Java™ on systems powered by the new IBM POWER6™ processor. It discusses memory-related performance and tuning, providing suggestions for tuning the Java heap and descriptions of various garbage collector configurations. It also describes thread-related performance tuning, touching on the AIX® threading model used by Java, as well as how dynamic logical partitions are exploited. The paper also provides a description of improvements to startup times, general Java coding guidelines and exploitation of various POWER6 hardware features including simultaneous multithreading and decimal floating-point hardware.

This paper is targeted towards and will benefit developers, technical support engineers, and system engineers who have basic knowledge of Java and tunable parameters.

Introduction

System engineers and Java practitioners are offered lots of choices when it comes to the availability of Java tuning guides ranging from the most basic to the more complex Java diagnostics guides. Often, the proliferation of such guides leads to confusion for those deciding on the right configurable settings for their Java environments.

This paper targets everyone who is willing to put a little bit of effort to attain big rewards in terms of reliability, availability, and serviceability (RAS) of their Java applications and middleware. More specifically, this paper provides a checklist of memory-tuning parameters, describes AIX and Java threading models, provides general Java coding guidelines, and describes POWER6 processor-specific hardware exploitation performed by various versions of the IBM SDK for Java.

The POWER6 processor is the latest IBM offering in the Power Architecture™ technology used in System p™ and System i™ servers. Optimized for very high clock rates, the POWER6 processor retains many successful elements of the previous POWER™ processors while also adding a number of new features. Among the features found in previous processors and retained in POWER6 are dual cores per chip, two-core simultaneous multithreading, and multiple page sizes. Changes include a doubling of the clock rate as compared to the previous generation (while at the same time keeping constant or reducing the cycle count latencies of most instructions), decimal floating-point hardware support, cache improvements, higher data bandwidths, support for the AltiVec™ SIMD instruction set, and better processor-to-processor communication.

References for each topic can be found in the “Resources” section at the end of this document.



Prerequisites

Our target audience should be familiar with Java on the AIX platform, Java technologies including classes, objects, garbage collection, Java heap, just-in-time compilation, and general object-oriented programming concepts.

Memory-Related Performance and Tuning

Memory behavior is invariably one of the most important components in program performance tuning. Java is no exception in this regard.

Tuning the Java Heap for Performance on AIX

Java programs run in a managed runtime environment provided by the Java virtual machine (JVM). It needs to co-exist in the same AIX process with the JVM itself and possibly other dynamically loaded native modules. One of its most important components is called the Java heap, an area of memory from which Java objects are allocated, referenced, and garbage-collected when they are not needed anymore. The size of the Java heap has a direct impact on the performance of Java programs, but it is competing for memory resources with other native memory consumers (module texts, thread stacks, native heap, shared memory, and mmap-ed files). Tuning the Java heap to an appropriate size could arise as a necessary step in typical middleware setups for performance and/or functionalities.

If an unnecessarily big Java heap is specified, memory resources could be wasted, and other modules could be denied memory requirements. If the Java heap becomes nearly full, and very little garbage is to be reclaimed, requests for new Java objects might not be satisfied quickly because no space is immediately available. If the heap is operated at near-full capacity, application performance might suffer as garbage collection activities are frequently initiated; and, if requests for more heap space continue to be made, the application may end up receiving an `OutOfMemoryException`, which results in JVM termination if the exception is not caught and handled. At this point, the JVM produces a "jvadump" diagnostic file.

Although the IBM SDK for Java can adaptively grow the Java heap to meet a Java program's memory requirements, it is not a good practice for optimal performance since the growing process usually results in many garbage collection activities. There are command-line options to both shape the Java heap and tune its size for performance. The **-verbose:gc** option can be used to request detailed information about garbage collection activities. Furthermore, the **-Xverbosegclog:** option can be used to capture the detailed information in files. An appropriate Java heap size can usually be derived from analyzing the frequency of garbage collection, its duration, and the Java heap occupation information. With Java heap size information in hand, the **-Xmx** option can be used to specify the maximum Java heap size allowed while the **-Xms** option is used for the initial Java heap size. For performance reasons, it is recommended that **-Xmx** and **-Xms** specify the same size for the default throughput-oriented garbage collection policy to avoid the Java-heap-growing process.

With a 32-bit Java program on AIX where the address space is broken up into 16 segments of 256 MB each, the AIX address space model being used by the program determines how those segments can be

used. Specifically, the address space model determines how many segments are available to the native heap and how many segments are available for other purposes. The AIX address space model is controlled by the **LDR_CNTRL** environment variable. The IBM SDK for Java uses the large program support in AIX by automatically setting the **LDR_CNTRL** environment variable as appropriate for allocating the Java heap. In almost all instances, the automatic settings of **LDR_CNTRL** will be adequate. In certain instances, however, a user might want to override the automatic settings and manually use an explicit **LDR_CNTRL** setting. It should be noted that, if an explicit setting is used, the SDK does not attempt to change it. In that case, Java heap sizes that are in conflict with the user's setting may not be satisfied. Adjustments to either or both of them would be required.

Garbage Collection

This section describes the garbage collector (GC) technologies available in the IBM SDK for Java 5.0 and 6.0. Detailed information and some useful in-depth discussion can be found in the reference material. To take full advantage of our technology, you need to understand the characteristics of your application (e.g. object mortality rate, heap size requirements), establish a performance target, and choose a configuration suitable for your application. Re-evaluation is also recommended when moving to a different system or upgrading to a new Java version.

GC Configurations

In response to a variety of performance requirements from client applications and benchmarks, IBM SDK for Java 5.0 and 6.0 have implemented a number of high-performance garbage collectors to provide a broad selection of approaches and strategies for garbage collection. The objective of the framework is to give clients the flexibility of selecting a garbage collector suitable for their applications in a given environment. In the following sections, the characteristics of a number of GC configurations are described, along with their advantages and drawbacks. Hints and tips are presented with respect to selecting policies. Each configuration is specified using **-Xgcpolicy:<policy name>**.

-Xgcpolicy:optthruput (the default collector)

This configuration is the default GC policy. It uses IBM SDK's base global collector, which is a mark-and-sweep collector that executes mark, sweep, and an optional compaction phase in sequence, with the help of multiple GC threads to accelerate their performance. The mark phase locates all the live objects while the sweep phase removes all unmarked objects and returns the freed space to a free list from which future memory allocations will be served.

Frequent mark-and-sweep collections may cause heap fragmentation. To address this problem, the compactor moves the objects and merges the scattered small free regions into several large, contiguous free regions. Given the overhead of moving live objects across the entire heap, compaction is triggered only through user command-line options or when heuristics decide it is necessary. For details on signs of fragmentation and tips to reduce it, refer to the section "Heap Fragmentation".

This configuration is typically used for large-heap applications when application throughput, rather than maintaining short pauses, is the chief performance goal. Because the application program stops while the global collection occurs, long pauses proportional to the heap size can be observed. If your application cannot tolerate long GC pause times, consider using optavgpause as an alternative GC policy.

-Xgcpolicy:optavgpause

This configuration uses the global collector with concurrent mark-and-sweep collection enabled. It is concurrent in the sense that a portion of the mark and sweep task is delegated to application threads. Therefore, some garbage collection work can be done concurrently while the application is running. As a result, individual pause times for a final collection phase are greatly reduced, at the cost of slowing down throughput to 5-10%, depending on the application.

This configuration is typically used for applications that must respond quickly to events, so shorter pause times are more important than achieving the fastest throughput. It is important to note that pause time contributes to, but is not equivalent to, response time, which measures how quickly an application responds to incoming requests. Improving response time often requires an extensive analysis of all the components and behaviour in the application, including garbage collection.

-Xgcpolicy:gencon

While the global collector, with optional concurrent behavior, works well for a wide variety of workloads, applications with many short-lived objects might benefit from gencon, which employs an incremental and localized approach. This configuration uses both the generational and global collectors. The generational collector splits the heap into two areas, a new space and an old space. The new space itself is partitioned into an allocate area and a survivor area. Objects are always allocated into the allocate area, which is scavenged when it becomes full. After a scavenge, live objects are copied into the survivor area and the two areas switch roles. When the number of times that an object is copied reaches a threshold (known as tenure age), it is promoted to the old space. When the old space becomes full, a global collection is triggered. Concurrent-mark but not concurrent-sweep is used for the old space.

This configuration is suitable for workloads with high object-creation and mortality rates, as is typical of many transactional applications. As short-lived objects are purged regularly in the new space without walking the entire heap, pause times are shorter than with optthruput. Another noticeable benefit is reduced heap fragmentation and improved memory locality as a result of incremental compaction of the new space.

-Xgcpolicy:subpool

In the default collector, object allocation requests, if not filled on a thread local heap, will go to a single list shared by all threads. A global heap lock is used to ensure that only one thread at a time can access the list and allocate an object. On large SMP machines, when many threads are allocating from the heap, there can be noticeable contention on the heap lock. The subpool technology, using multiple free lists of different sizes, was developed to relieve the heap lock contention. This configuration is suitable for optthruput-type workloads where there are many threads allocating in a large (> 1 GB) heap. It is not typically beneficial to run this collector with fewer than 16 CPUs.

Heap Fragmentation

Heap fragmentation is used to describe a state when the available memory on the heap, although large by total amount, consists of many small chunks of free space so that none can satisfy allocation requests above a certain size.

There are two ways that you can tell if fragmentation has become a problem:

- Examine the log generated by **-verbose:gc**. The most common sign is that allocation failure occurs even though the heap is not 100% full.
- Examine the average size of thread local heaps (TLH) generated by **-Xtgc:freelist**. As fragmentation of the heap can prevent the thread from getting large TLHs, a TLH fills quickly, causing the application thread to come back frequently to the heap for a new one but can only get a TLH of a small size. In this situation, you may also notice in profile data that heap lock acquisition consumes a high amount of CPU time.

To reduce heap fragmentation, the IBM SDK employs various techniques, most of which are adaptive with optional tuning parameters. Here are some tips on how to tackle heap fragmentation problems using command-line options:

- If `gencon` yields acceptable performance, use this GC policy to avoid heap fragmentation.
- If `gencon` is not an option and heap fragmentation is severe, consider starting with a very small heap and then allowing the garbage collector to expand. This will trigger early compaction, when the heap is small and the cost of compaction is lower.
- When using `optthruput`, if `System.gc()` is explicitly called from the application code, you may consider using option **-Xcompactexplicitgc** to automatically trigger a compaction when `System.gc()` is called.
- Option **-Xcompactgc** triggers a compaction after every GC. As compaction is an expensive operation, this option should be used with caution.

The above GC options are explained in the IBM SDK 5 Diagnostics Guide.

It is worth mentioning that IBM SDK 5 has taken great strides in improving fragmentation concerns over previous IBM SDK releases. This is due to significant re-architecting and removing the concept of pinned objects. (These were objects that could not be moved and therefore contributed greatly to fragmentation.)

Large Pages

The physical memory in a computer is divided into pages that are managed by the operating system. The operating system allocates the pages to the tasks running on the system. When pages are in short supply, the operating system writes some memory pages to a special area of the hard drive called the paging space. Through a combination of the physical memory in the system and the paging space on the



disk, the operating system supports a virtual memory larger than the physical memory and transparently gives each task the appearance of a large virtual memory space.

The Power Architecture originally supported only a 4 kilobyte (KB) page size. The POWER4™ processor introduced support for a new 16 megabyte (MB) page size. The 16 MB pages required special setup in the operating system and had a number of restrictions on their use. They were in a special reserved area of the physical memory, and that area was only available to tasks specifically requesting 16 MB pages and authorized to use those pages. The 16 MB pages were pinned in memory and could not be paged out to the disk paging space.

More recently, the IBM POWER5+™ processor introduced support for additional page sizes such as 64 KB pages. The new 64 KB pages function identically to the original 4 KB pages - they don't require any special setup, they are allocated from the same unreserved physical memory that is used for 4 KB pages, they are available to all tasks, and they can be paged out to the disk paging space.

The POWER6 processor supports all of the page sizes supported by the POWER5+ processor. Java supports all of the page sizes through the **-Xlp** option. By default, Java uses only 4 KB pages. When the **-Xlp** or **-Xlp16m** option is specified, Java will switch to 16 MB pages for the Java heap, assuming the necessary operating system setup and authorization has been done. With the **-Xlp64k** option, Java will use 64 KB pages for the Java heap.

Using large pages can significantly improve an application's performance because of the hardware efficiencies associated with larger page sizes. This is particularly true of applications that allocate and use very large contiguous areas of virtual memory. Many Java applications fall into that category, notably if they allocate a large Java heap. The 16 MB and other very large pages are somewhat restrictive in their setup and usage, and are intended for very high-performance environments, while the medium-sized 64 KB pages are general-purpose and many workloads will see a benefit by using them rather than the default 4 KB pages.

To demonstrate the potential performance improvement by using large pages for the Java heap, a workload was run on a POWER6 processor with default 4 KB pages, using **-Xlp64k** so that 64 KB pages were used for the Java heap, and using **-Xlp16m** for a Java heap with 16 MB pinned pages.

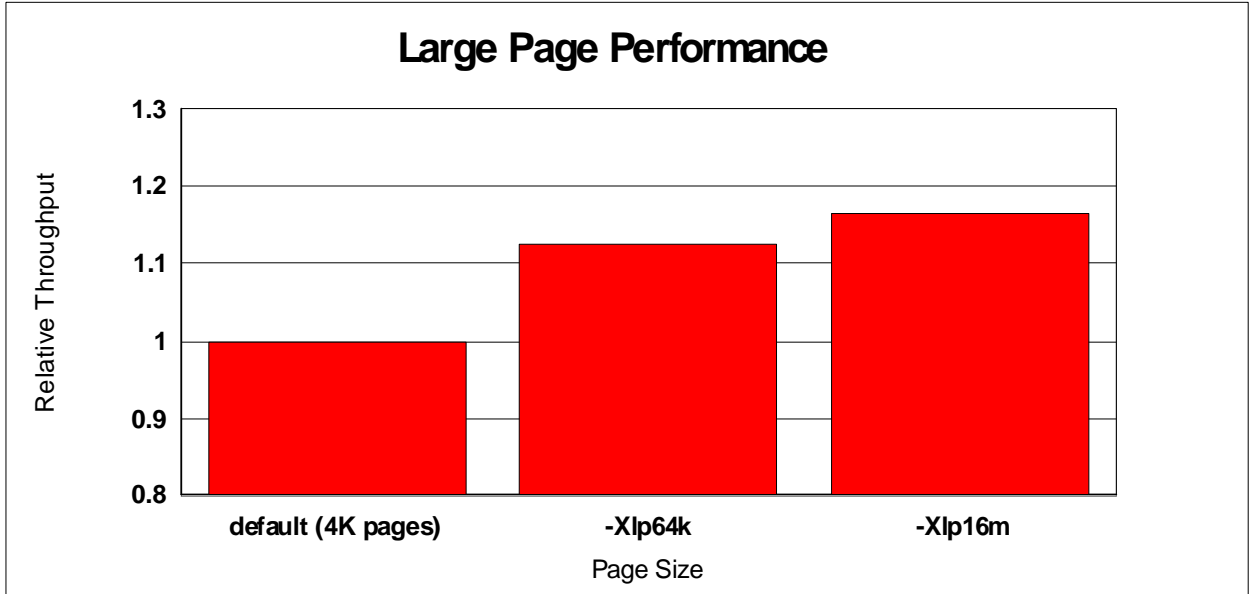


Figure 1- Large Page Performance

These results show a substantial boost in performance using 64 KB pages, and an additional smaller boost by using 16 MB pages. Our recommendation is to always use the **-Xlp64k** Java option under normal circumstances, and performance will often be improved by 6-10% or more. Where it is important to achieve the highest possible performance, the necessary operating system setup can be performed and the **-Xlp16m** Java option can be used and will often improve performance by 8-16% or more.

It is also possible to use the large pages for memory areas other than the Java heap. These other memory areas include the program text (machine instructions), data area (all other data outside of the Java heap), and stack. Under the AIX operating system, one mechanism for using large pages for the other data areas is through the use of environment variables, such as in this example:

```
export LDR_CNTRL="TEXTPSIZE=64K@DATAPSIZE=64K@STACKPSIZE=64K"
```

Running Java after setting the environment variable as in this example will cause 64 KB pages to be used for all of the text, data, and stack areas. More details on changing the page sizes used by programs are available in the AIX documentation.

Running under the AIX operating system, Java generates code for compiled Java methods in the program data area. In applications with many Java methods that generate a large amount of compiled code for those methods, using large pages for the program data area can provide a noticeable boost in performance. In our experiments, we have seen up to a 10% improvement in performance by using 64 KB pages for the program text/data/stack when running large Java applications.

Thread-related Tuning

Threading Model

The following section describes the threading model followed by Java Standard Edition 6 (Java 6) on AIX, and shows the user how to configure for performance tuning.

AIX Threading Models

On AIX, the threading model places user threads on top of virtual processors, which execute on top of kernel threads. Each virtual processor can be thought of as a virtual CPU available for executing user code and system calls. If the user threads need to access kernel services such as system calls, they will be serviced by the associated kernel threads.

On AIX, multithreaded applications have a choice between two different threading models:

- **1:1 Thread Model:** Each user thread is bound to a virtual processor and linked to one kernel thread. This is called *system scope* because threads are directly scheduled with all the other user threads by the kernel. The virtual processor is not necessarily bound to a real CPU, unless it is explicitly configured that way.
- **M:N Thread Model:** Several user threads can share the same virtual processor or the same pool of virtual processors. This is called *local* or *process scope* because threads are not directly scheduled with all the other threads by the kernel scheduler. The threads library handles the scheduling of user threads to the virtual processor and then the kernel will schedule the associated kernel thread. This model was implemented in AIX 4.3.1, where it is the default model. As of AIX 4.3.2, the default is to have one kernel thread mapped to eight user threads.

Java 6 supports AIX 5.2 and above. The threading model that Java follows is 1:1.

Threads Created by Java

The different components in Java create a series of threads, each with a distinctive purpose. There is no direct control for the user over such threads, except where stated explicitly.

RAS	Reliability, availability, and serviceability. Writes trace buffers out to a file as they become filled, and it creates snap dumps in the event that a snap is requested.
Port library	Processes the requests made by asynchronous signals. For example, javacore generation upon receiving a SIGQUIT.

Garbage collection	Garbage collection uses helper threads. This depends on the policy, and the user can control the number of threads.
Application threads	These depend on the nature of the Java application being executed. There is a one-to-one mapping between Java threads and operating system threads.
Compilation	Just-in-time compilation thread.

The garbage collector spawns several helper threads. This depends on the garbage collection policy that is selected. (See “[Garbage Collection](#)” above for further information.) A platform with n processors will have n-1 helper threads by default, and on single CPU boxes, no helper threads run. These threads work along with the main garbage collection thread during parallel mark phase, parallel bitwise sweep phase, and parallel compaction phase. The number of garbage collection helper threads can be controlled with the **-Xgcthreads** option. Passing the **-Xgcthreads1** option to Java results in no helper threads at all, which disables parallel operations at the cost of performance. There is no performance gain by setting **-Xgcthreads** to more than the default setting (n-1). The Diagnostic Guide in the [resources](#) section recommends not to do so, except when alleviating mark-stack overflow errors.

Scheduling Scope

On AIX 4.3.1 and later versions, it is possible to control the contention scope of a process with the AIXTHREAD_SCOPE environment variable. Possible values are P (for *process scope* or M:N model) and S (for *system scope* or 1:1 Model). The default value is P. (See “AIX Threading Models” above.)

However, Java 6 sets the scope explicitly to *system scope*. Therefore, this environment variable will have no effect on the scheduling.

Scheduling Policy for Threads

A thread with *system scope* has its scheduling policy tied to the system scheduling. This implies that any change in its scheduling policy has to be made at system level. There are several different scheduling policies for threads, as follows:

- SCHED_FIFO: After a thread with this policy is scheduled, it runs to completion unless it is blocked, it voluntarily yields control of the processor, or a higher-priority thread becomes dispatchable. Only fixed-priority threads can have a SCHED_FIFO scheduling policy.
- SCHED_RR: When a SCHED_RR thread has control at the end of the time slice, it moves to the tail of the queue of dispatchable threads of its priority. Only fixed-priority threads can have a SCHED_RR scheduling policy.
- SCHED_OTH: This policy is defined by POSIX Standard 1003.4a as implementation-defined. The recalculation of the running thread's priority value at each clock interrupt means that a thread may lose control because its priority value has risen above that of another dispatchable thread.

- SCHED_FIFO2: The policy is the same as for SCHED_FIFO, except that it allows a thread that has slept for only a short amount of time to be put at the head of its run queue when it is awakened. This time period is the affinity limit (tunable with `schedo -o affinity_lim`). This policy is only available beginning with AIX 4.3.3.
- SCHED_FIFO3: A thread whose scheduling policy is set to SCHED_FIFO3 is always put at the head of a run queue. To prevent a thread belonging to SCHED_FIFO2 scheduling policy from being put ahead of SCHED_FIFO3, the run queue parameters are changed when a SCHED_FIFO3 thread is enqueued, so that no thread belonging to SCHED_FIFO2 will satisfy the criterion that enables it to join the head of the run queue. This policy is only available beginning with AIX 4.3.3.
- SCHED_FIFO4: A higher-priority SCHED_FIFO4 scheduling class thread does not preempt the currently running low-priority thread as long as their priorities differ by a value of 1. The default behavior is the preemption of the currently running low-priority thread on a given CPU by a high-priority thread that becomes eligible to run on the same CPU. This policy is only available beginning with AIX 5L™ Version 5100-01 + APAR IY22854.

Time Slice

The processor time slice is the amount of time a SCHED_RR thread can absorb before the scheduler switches to another thread at the same priority. This is not a guaranteed amount of processor time, but the longest time that a thread can be in control before it faces the possibility of being replaced by another thread. You can use the *timeslice* option of the *schedo* command to increase the number of clock ticks in the time slice by 10-millisecond increments. See the *schedo* reference for further information.

Spin Loop Time

If an application is running on an SMP system and a user thread cannot acquire a mutex, by default it will attempt to spin a few times. However, depending on the nature of the application, it could be the case that the mutex is available within a short amount of time. In such cases, it may be worthwhile to adjust the spin to try for a longer period of time.

The SPINLOOPTIME environment variable controls the number of times (number of spins) that the system will try to get a busy mutex or spin lock without taking a secondary action, such as calling the kernel to yield the process. This control is intended for multiprocessor systems, where it is hoped that the lock being held by another actively running thread will be released.

A general tuning rule is that if more CPUs are added and the performance goes down, this can indicate a locking problem. If locks are usually available within a short amount of time, you may want to increase the spin time by setting this environment variable. The default value of this parameter is 40 and the value must always be a positive value.

Maxspin

The MAXSPIN kernel parameter affects spinning in the kernel lock routines. It represents the number of times to spin on a kernel lock before sleeping. It can be modified using the *schedo* command.

Yield Loop Time

The YIELDLOOPTIME environment variable controls the number of times that the system yields the processor when trying to acquire a busy mutex or spin lock before actually going to sleep on the lock. The processor is then yielded to another kernel thread, assuming there is another executable one with sufficient priority. The value must be a positive number, the default being 0.

AIX Performance Tools

The AIX Performance Tools is a set of packages that simplify the administrator task of tuning AIX performance. The IBM Redbooks® publication, *AIX 5L Performance Tools Handbook*, describes a series of concepts and tools related to AIX performance tuning. Among other commands, AIX Performance Tools provide the *schedo* (previously known as *schedtune*) command, which can be used to set or display current or next boot values for all scheduler-tuning parameters. It resides under */usr/bin/* and can be installed from the *bos.perf.tune fileset*. Only the root user is authorized to execute it. For example, *schedo* can be used to change thread priorities and aging, modify the time slice, lock tuning, etc. Misuse of this command can cause performance degradation or operating-system failure. The Performance Tools Handbook reference (see [resources](#)) is a good guide that will show the use of *schedo* to change system parameters.

Exploiting Dynamic Logical Partitions with Java

A logically partitioned (LPAR) system is one that has been logically divided into *n* systems, where each partition runs a separate operating system (OS) image (AIX, Linux®, OS/400®) and behaves as a separate system. In addition, the OS level may differ across partitions. Dynamic LPAR increases the flexibility of LPAR'ed systems by allowing selected resources to be added to and deleted from partitions without requiring a reboot of the system or affected partitions. Dynamic LPAR depends on the use of a Hardware Management Console (HMC) to control the placement of resources across partitions, a distributed framework to instruct the OS to integrate or isolate resources, and firmware support to electronically isolate and integrate resources. The movable resources among running partitions include individual CPU processors, physical memory regions, and I/O adaptor slots.

Dynamic LPAR systems pose some challenges to typical long-running middleware's, mainly for performance reasons, in adapting to the potentially changing nature of the underlying partition. The IBM SDK for Java exploits dynamic LPAR in a few aspects at different levels.

1. The garbage collector can dynamically change its behaviors on resource moves in the partition. The soft Java heap limit (which can be controlled by the **-Xsoftmx** option) can expand or shrink as the physical memory resource is moved in or out of the partition. The number of worker threads used in garbage collection (which can be specified by the **-Xgcthreads** option) can

increase or decrease dynamically as the processor resource in the partition is added or taken away.

2. Users can inform the SDK of the outlook on partition resource changes. If Java applications run on a single CPU LPAR and no CPU is ever dynamically added to that LPAR while those Java applications are running, performance can be improved by exporting the following environment variable: export **NO_LPAR_RECONFIGURATION=1**. Do not export this environment variable unless all of the following usage points can be guaranteed:
 - Running in an LPAR.
 - The LPAR has one CPU.
 - The LPAR will never be dynamically reconfigured to add more CPUs while Java applications are running.
3. To enable applications to respond to dynamic LPAR events, the SDK for Java includes specific IBM extensions to `java.lang.management` that provide a Java interface to query various LPAR-specific information and listen for events indicating that the JVM's logical partition has been dynamically altered. The API for monitoring memory management of the JVM is supported through interface `com.ibm.lang.management.MemoryMXBean`, while the API for monitoring the operating system, in particular, the entitled CPU processing capacity and the available processor number, is through interface `com.ibm.lang.management.OperatingSystemMXBean`.

Among other methods, the IBM `MemoryMXBean` provides `getMaxHeapSize()` and `setMaxHeapSize(long size)` for applications to dynamically adjust the soft Java heap limit of the JVM in response to physical memory resource changes in the partition. For example, when physical memory resource is reduced, it might be beneficial for performance to lower the Java heap limit by calling `setMaxHeapSize` with a smaller value than returned from `getMaxHeapSize`, and to prevent the partition from slipping into constant paging, a situation due to shortage of physical memory.

The IBM `OperatingSystemMXBean` is a `NotificationEmitter`. Among others, it can broadcast three specific IBM classes of notification: `AvailableProcessorsNotificationInfo`, `ProcessingCapacityNotificationInfo`, and `TotalPhysicalMemoryNotificationInfo` (all in `com.ibm.lang.management` package). Applications can register event listeners to receive and respond to these notifications upon resource changes in the underlying partition, through the API `addNotificationListener`.

Startup Performance

Reducing Java Application Startup Time

Since just-in-time (JIT) compilation of Java bytecodes occurs at run time, its overhead can have a detrimental impact on an application's performance. The most affected scenarios are startup phases of big server applications, such as WebSphere® Application Server, because this is where the bulk of compilations takes place, and short running applications, because there isn't enough time to amortize the cost of compilation. The JVM from IBM employs a few techniques to reduce or hide the JIT compilation overhead:

1. The IBM JIT compiler uses an adaptive multi-level compilation technology where only the methods that get executed more often get compiled (based on an invocation counter). The execution of compiled code is continuously profiled by a dynamic sampling mechanism that kicks in approximately every 10 ms. As the JVM detects "hot" methods in the application, those methods may get recompiled at higher optimization levels. Thus, the JVM spends more compilation resources on code that has a bigger impact on performance.
2. JIT compilation is performed by a separate thread that works asynchronously to the application's threads. The benefit is obvious on a multiprocessor machine where the compilation can be carried out on a separate processor without stalling the Java threads.
3. For big server applications with a flat profile – i.e., thousand of methods without clear hot-spots - the effectiveness of the adaptive multi-level compilation is diminished. In such cases, to limit the JIT compilation overhead, the JVM identifies phases of execution for which there is an intense class-loading activity and reduces the initial optimization level for compilations performed during such phases.

All of the above techniques happen automatically without requiring user intervention. The goal is to improve the startup of applications without affecting their long runtime performance. However, sometimes the user prefers a fast startup or a short response time for its application, rather than long-term performance. The classic example of such a scenario is a developer who needs to start and test his/her application many times a day. Other common examples include short-lived and graphical/interactive applications. To address such requirements, the JVM provides the **-Xquickstart** command-line option. The savings come from three main directions:

1. Compilation decisions are delayed (a method needs to be invoked more times to transition from interpreted to compiled).
2. The optimization level for first-time compilations is reduced.
3. The mechanism of profiling bytecode execution (interpreter profiling) is turned off.

It is worth mentioning that **-Xquickstart** does not preclude the recompilation of Java methods. As methods get sampled by the dynamic profiling mechanism, they may be identified as hot-spots and recompiled at a higher level of optimization. This ensures that runtime performance is not duly sacrificed for the sake of startup time benefits.

Further startup time improvements can be achieved by turning off the sampling-based recompilation mechanism after a specified period. This can be accomplished with the **-XsamplingExpirationTime<n>** command-line option, where <n> denotes the number of seconds after which the sampling mechanism is turned off. This option eliminates recompilations, but does not prevent first-time compilations from happening because the latter rely on the invocation counting mechanism instead of sampling. Between the two options, **-Xquickstart** should always be preferred to **-XsamplingExpirationTime** when seeking a short startup time. The reason is that it has a greater potential for compilation time savings, while in general offering a better runtime performance.

The two command-line options described above have an additional benefit that is sometimes overlooked: they help reduce the CPU cycles burnt by the compilation thread. This topic is of special interest to S/390® clients who are billed based on the CPU cycles they consume.

As a last point, it is important to note that startup time of Java applications can be further improved by using the so-called “shared classes cache” and ahead-of-time (AOT) code. This topic will be detailed in the next section of this document.

Exploiting Java Shared Classes and Ahead-Of-Time (AOT) Compilation

Java programs are apt to be sluggish during their incipient phases. One factor contributing to that is the large number of classes loaded during the startup phase. The process of verifying and loading the classes makes up a significant portion of startup time. For typical long-running middleware, such as an application server, in which down time is very critical, the application needs to be recovered and started up quickly. In addition, middleware with multiple JVMs interacting with each other across address spaces consume storage space per JVM.

The idea of sharing classes helps mitigate both the storage consumption and the startup time. The real (auxiliary) storage consumed per JVM can be reduced by sharing classes between multiple JVMs. Shared Classes is the technology in the IBM SDK for Java that allows all system and application classes to be stored in a persistent dynamic class cache in shared memory, called the “shared cache”. Classes are verified and loaded into a shared and persistent location. Other JVMs loading the same class can directly use the class that is stored in the shared location. In addition to reducing memory consumption, class sharing can speed up the class-loading process. JVMs can search the persistent location for the preloaded classes, eliminating the need to load classes from scratch.

Another factor contributing to slow startup time is dynamic compilation time. The JIT compiler selectively compiles frequently executed methods into optimized native code. The time used to compile methods is added to total execution time since the JIT compiler operates while an application is being run. Since methods begin by being interpreted and the bulk of JIT compilations, with their inherent compile-time overhead, occur during startup, this phase generally tends to be sluggish. Startup performance can be improved by allowing native code to be run without the compilation overhead. The idea of AOT compilation can achieve this. Methods are compiled into native code ahead of time before the actual execution of a program. The precompiled native code can later be executed without incurring compilation overhead.

While traditional AOT compilers, which generate native code statically, lose the platform-neutrality of Java, the IBM SDK for Java employs AOT technology in a way that maintains platform-neutrality. Native code can be precompiled and generated dynamically while an application is run. After the code is generated for the first time, the code is stored into a persistent cache. Subsequent JVMs that want to execute the method can load and use the AOT code from the persistent cache without incurring the compilation overhead generally experienced for JIT-compiled native code. There is a small price to bind the AOT code into the JVM but it is miniscule compared to the time it takes to compile a method. Because of the benefit of optimized native code without the compilation overhead using this technology, middleware with critical downtimes can benefit from improved startup performance. Also, for users who are billed for CPU utilization, AOT code can be used to reduce CPU utilization since CPU resources are free from performing method compilations.

The **-Xshareclasses** option can be used to turn on both Shared Classes and AOT. The two technologies share the same persistent cache. The size of the cache can be set by using the **-Xscmx<x>** option. While Shared Classes reduces memory consumption, the AOT code increases the memory consumption. An expansion factor of approximately 10 times can be expected of compiled native code. Moreover, the AOT code is not shared in the same way that classes are shared with Shared Classes. Whereas multiple JVMs can directly use the classes that are shared in the shared cache, the AOT code cannot be directly used. The AOT code needs to be loaded into the local memory space and bound to each JVM that uses it. The benefit, as mentioned, is for startup performance purposes and reduced CPU utilization. The minimum and maximum amount of cache space for AOT code can be tuned by using the **-Xscminaot<x>** and **-Xscmaxaot<x>** options, respectively. The **-Xshareclasses:noaot** option can be used to allow Shared Classes but disable AOT.

To demonstrate the potential improvement generated by AOT code, Figure 1 captures startup time improvements of Eclipse running with AOT code. The results were gathered on a POWER4 4x1453 MHz and the measurements are representative of the kinds of improvements one would expect on a POWER6. The results show a 14% improvement with AOT code.

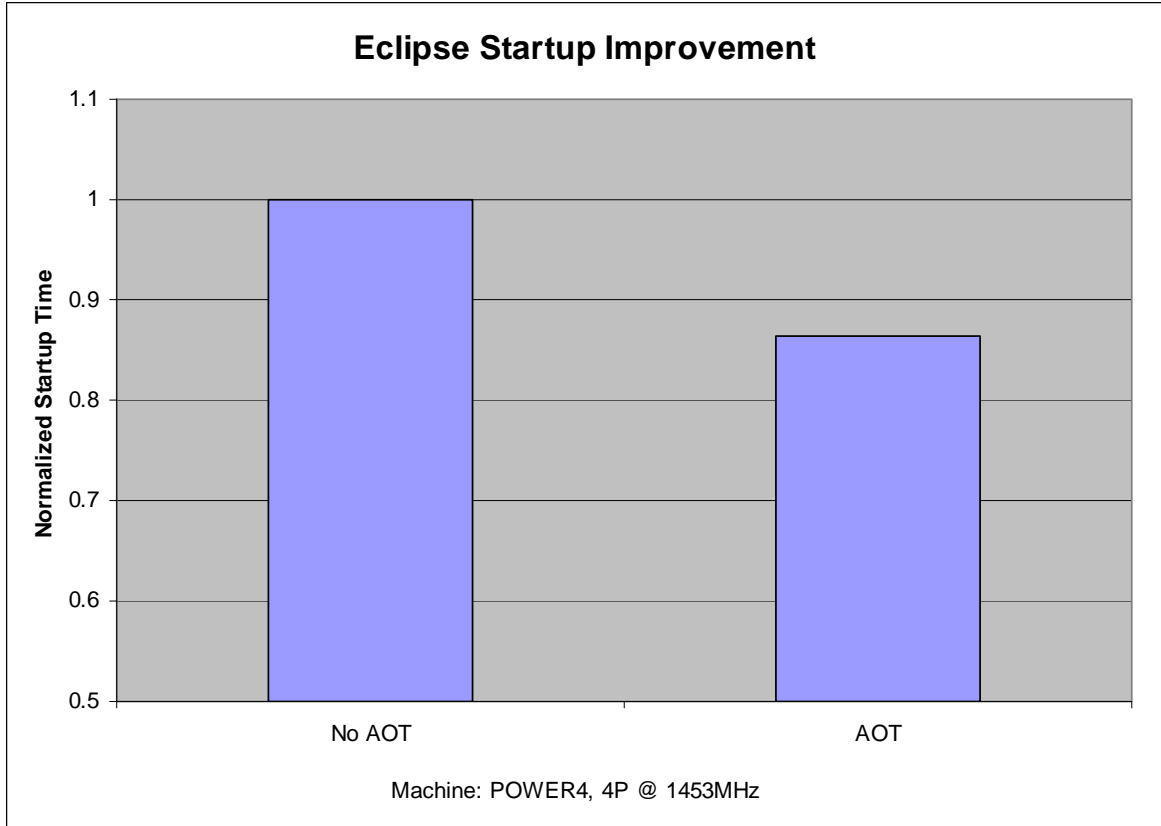


Figure 2 - Eclipse Startup Improvements

Java Coding Guidelines

In this section, we present a collection of Java coding guidelines based on our experience in analyzing and improving Java server performance from a VM and a JIT compiler perspective. Aimed at helping designers and programmers accelerate server application without compromising good design principles, these are recommended practices based on current technology, not strict rules.

Object Allocation (don't think GC can do everything for you)

The garbage collector can help improve locality by reordering objects such that objects that are expected to be accessed within a short duration of each other are adjacent in memory to each other. However, the garbage collector does not reorder fields within an object such that fields accessed within a short duration of each other are adjacent to each other; the order in which fields are stored within an object of a certain class is decided at class loading/initialization and does not change for the duration of the run. A user could increase the likelihood that hot fields would be stored adjacent to each other within an object by implementing classes that are as compact as possible in terms of number of fields (the fewer the number of fields, the higher the likelihood that frequently accessed fields can be stored adjacent to each other) and by re-factoring classes such that the number of fields accessed infrequently is kept to a minimum. If

there is a class C with a large number of infrequently accessed fields, one possibility could be to create a separate class D that encapsulates all the cold fields and have a single field in an object of class C that points at an object of class D. This would introduce extra overhead (indirection) in accessing the cold fields, but the benefits of improved locality for frequently accessed field(s) should outweigh this cost.

Another important issue that affects locality is the amount of object allocation performed by a program. Even if the program creates an object that is subsequently unused (i.e., never read) or used only for a very short duration, it may be necessary to allocate it on the heap and initialize it. Sometimes, the compiler can prove that such an object can be stack allocated by performing escape analysis, but it is also possible that this analysis does not have enough information at compile time to decide to perform a stack allocation instead of a heap allocation. In Java, the presence of dynamic class loading means that virtual calls might invoke methods that are not available for analysis at compile time. Further, it may be impractical to perform the exhaustive analysis required to prove that an object can be stack allocated given the constraints on compilation overhead in a dynamically compiled language such as Java. Even if an object is simply allocated and never accessed again, initialization of the object would mean that the memory is brought into the cache, thereby evicting some other object (which may have been very frequently accessed). So, users should try to reduce the number and the size of allocations in their programs as much as possible. In order to improve the likelihood of stack allocating an object, care should be taken to avoid storing the object into a static or instance field.

Classes, Methods and Fields

- It is usually beneficial to keep the class hierarchy as simple as possible. Because in general interface calls are more expensive than other calls, excessive use of interfaces is something that could affect performance significantly. It is also a good idea to consider using an abstract class instead of an interface where feasible.
- Try to declare static and instance fields as private and final where applicable. Examine if it is possible to only write to such fields in constructors, since this would allow the compiler to consider these fields immutable and optimize them more aggressively.
- The compiler can optimize code in a better manner if important calls, static and instance field accesses can be validated and changed to a direct reference (resolution) during compile time. This might not be the case if all the important code is in a method that is invoked a very few number of times during a run (e.g., if all the code is in 'main'). In fact, earlier versions of the JVM (e.g., Java 5.0) might not even compile methods called only once, and this might result in the performance-critical code being interpreted.
- Declare parameters to be of a class type that is either final or at least a class type that is of a refined type (i.e., not 'Object' or some abstract/interface class) where possible. This would allow the compiler to devirtualize (and possibly inline) without a virtual guard.
- From both a performance and a design perspective, it is usually a good idea to factor the code such that the work is distributed among several small methods as opposed to creating a large

and complex method that performs almost all the work. Use final or private methods where applicable, as this practice allows more aggressive optimization.

- Because of the additional levels of abstraction and indirection required, reflection tends to have a higher cost than normal code. Therefore, excessive use of reflection, especially in a loop, is not recommended.

Loops and Controls

- The compiler can perform more aggressive optimization when a loop is well behaved. When constructing a loop, try to make it as compact as possible; consider keeping the upper bound loop invariant and use single increment value on the induction variable across all paths if possible. It is also a good practice to examine if there is any cold code mixed with hot code in a loop. If so, try putting them in two separate loops.
- Using the Java class library method `System.arraycopy` is advised instead of writing your own array copy code because `System.arraycopy` has been well optimized by the compiler.
- As processing exceptions and handling finally blocks are prohibitively expensive, avoid using them on the commonly executed path. Exceptions should only be used in the case of handling abnormal errors.
- It is a common practice to use tracing and logging code to collect runtime information for a middleware application. As such code is rarely executed in a production environment, it has the potential to increase memory footprint and reduce performance of a system. Therefore, it is important to provide a mechanism to easily turn off or eliminate the code from the production build.

Threads and Locks

Synchronization is often used in multi-threaded programs to ensure thread safety of the application. Before writing synchronization code, it is necessary to understand the overhead of synchronization.

The main overhead is the cost of acquiring and releasing monitors. When a lock is contended by multiple threads, all but one will have to wait for the thread owning the monitor to finish its task and release the lock, which means more time spent in the JVM and kernel-locking routines and less time in the program code. Even when there is little contention and the synchronized code region is very simple, as the Java Memory Model mandates that caches be invalidated when a lock is acquired, and flushed before the lock is released (which implies that some memory synchronization instructions have to be used), this overhead cannot be fully eliminated, thus making synchronization an inherently expensive operation. Another overhead is the code executed inside the synchronized block. As only one thread can execute the

synchronized block at a time, the longer a thread holds a lock, the more likely other threads will have to contend for the lock.

IBM SDK has carefully designed the implementation of monitors to minimize the overhead of locking. The compiler also employs various optimization strategies, such as lock coarsening and special locks to further reduce the cost of synchronization. Although the performance of synchronization has been improved significantly, it can still entail a performance penalty if used excessively without much premeditation. Therefore, we recommend the following measures to avoid unnecessary lock contention.

- Without compromising thread-safety, always check if synchronization can be avoided. Be aware of cases when synchronization is not needed. Carefully analyze the data to see if you can restrict the data to a single thread or use `java.lang.ThreadLocal` to maintain per-thread data.
- Avoid unnecessarily synchronizing on the same lock when you can use different locks for accessing different data.
- Make conscious choice when using objects and methods from the class library or middleware application code. Some use synchronization and may potentially introduce lock contention.
- To reduce the hold time of a lock, try to make synchronized blocks as short as possible. Move thread-safe operations out of the synchronized block.
- If you have some consecutive synchronized blocks locking on the same object, consider coarsening the lock by guarding all the code under one lock. This will reduce the number of acquires and releases.
- Avoid synchronizing static methods, if possible.

Using the `volatile` keyword has been considered by some programmers as a cheap and easy way to ensure atomic access to fields. This is no longer true now that JSR 133 significantly strengthens the semantics of `volatile`. The stringent rules inhibit the JVM from performing any optimization on `volatile` variables. As a result, `volatile` is no cheaper than using `synchronized`. In some cases, it is even more expensive if contention occurs. Therefore, we strongly recommend against using the `volatile` keyword

POWER6 Hardware Feature Exploitation

Simultaneous Multithreading

Simultaneous multithreading (SMT) is where multiple threads can execute on the same processor core at the same time. This means that there are multiple logical processors within each physical core on the chip. Each logical processor independently executes a separate stream of instructions and has its own set of context (program counter, registers, etc). The other parts of the core, however, are not duplicated and so there is only one set of functional units on which to actually execute instructions. This means that SMT is not as powerful as using separate cores, but it does offer some substantial performance benefits. When any functional units on a processor core are not being used by one thread, they can be utilized by another thread. When one thread is stalled by events such as a cache miss, the other threads can still continue execution.



The POWER4 processor first introduced dual processor cores on the same chip. The POWER5™ processor then introduced 2-core SMT within each core. Each POWER5 chip is therefore four logical processors, with each of the two processor cores capable of running two threads in SMT mode. There are some restrictions in the SMT on a POWER5, notably that instructions can only be dispatched (put into queues waiting for execution on a functional unit) from one thread during each cycle. SMT on the POWER5 processor has a performance benefit of up to about 40 percent. That is, having two threads execute on the same processor core with SMT will give as much as 1.4 times the instruction throughput of a single thread executing on the same core without SMT.

Similar to the POWER5, the POWER6 processor also has two processor cores per chip and has 2-core SMT within each core. Unlike the POWER5, however, the POWER6 processor can dispatch instructions from both threads in the same cycle. This improved SMT on POWER6 has a performance benefit of up to about 55 percent.

SMT support is generally completely transparent to a user. The Java compiler automatically makes use of SMT support by default and does not have any options controlling SMT. SMT can be queried and controlled with the AIX operating system command *smtctl*.

Decimal Floating-Point Unit Exploitation

The Decimal Floating-Point Unit (DFU) is a new accelerator that is included in the POWER6 processor core. Its primary purpose is to increase the performance and accuracy of financial transactions by providing a hardware mechanism for performing decimal radix arithmetic. Typical financial transactions involve many decimal operations and require exact decimal arithmetic. Decimal arithmetic cannot be directly implemented by current binary floating-point hardware because of problems with exact representation. For example, the decimal 0.1 does not have an exact representation in binary floating-point. While attempts have been made to emulate decimal arithmetic using binary fixed-point integers, the overhead and performance problems with doing so outweigh the benefits of the representation and supporting arithmetic.

The POWER6 processor is the first commercial hardware implementation of the new IEEE754R floating-point standard, which defines decimal floating-point formats, encodings, and arithmetic operations. The POWER6 DFU architecture adds 54 new instructions supporting the 32-bit storage format, as well as full 64- and 128-bit decimal arithmetic. Java 6 supports 64-bit decimal floating-point, and does so using the `BigDecimal` class.

Java 6 BigDecimal

The Java 6 `BigDecimal` class (rooted and identical to the Java 5 `BigDecimal` class), is significantly different from its Java 1.4.x predecessor. In addition to being able to represent the entire range of decimal floating-point numbers (now only restricted by a 32-bit signed integer scale), the Java 6 class provides more constructors, a number of new methods including, but not limited to, division, integral division, remainder, and `pow`. One of the more significant additions was the notion of a `MathContext`

object. A `MathContext` is used to specify both the precision of a `BigDecimal` arithmetic result and a `RoundingMode`, which determines the algorithm to be used for rounding the result.

64-bit Decimal Floating-Point in Java 6

Java 6 automatically exploits the POWER6 DFU and executes `BigDecimal` functions in decimal floating-point hardware. The 64-bit decimal floating-point format represents positive and negative `BigDecimal` values that have a maximum precision of 16 (i.e., 16-digit-long coefficients) and contain an exponent in the range -398 to 369 inclusive.

64-bit Decimal Floating-Point Coding Suggested Practices

`BigDecimal` and the POWER6 DFU are both optimized for `BigDecimal` arithmetic operations that use `MathContext.DECIMAL64`. `MathContext.DECIMAL64` specifies 16 digits of precision, a `HALF_EVEN` rounding algorithm and is the IEEE754R default for 64-bit decimal floating-point arithmetic. Sixteen digits of precision should suffice for most financial transactions. Should the situation require a result with a particular scale, we suggest rescaling using the `BigDecimal.setScale(int n, RoundingMode rm)` function.

The following snippet of code reads in a dollar value from a database, applies a tax, and prints the result as a “dollars and cents” value to the console:

```

...
BigDecimal taxRateBD = new BigDecimal("1.232"); // create tax
String initAmt = ... ; //read from db (135.23)
BigDecimal initAmtBD =
    new BigDecimal(initAmt); //create the BigDecimal

BigDecimal taxedAmtBD =
    initAmtBD.multiply(taxRateBD, MathContext.DECIMAL64); //apply tax

BigDecimal finalAmtBD =
    taxedAmtBD.setScale(2, RoundingMode.HALF_EVEN); //final amt(166.60)

System.out.println(finalAmtBD); //write out
...

```

Preliminary Performance Results

A modified version of `jtTelco`, a short and simple IBM test that captures the essence of a telephone company billing application, runs close to 16% faster when run on a POWER6 core with the DFU enabled, relative to an identical run on a POWER6 processor DFU disabled. It also runs approximately 45% faster than on a comparable POWER5 processor.

Other `BigDecimal` microbenchmarks record speedups of decimal floating-point hardware operations summarized below. The reported speedups are for runs on a POWER6 with the DFU enabled relative to identical runs on a POWER6 with the DFU disabled.



Microbenchmark	Speedup
BigDecimal Constructors	Up to 5X speedup in constructing BigDecimals from Strings and primitive data types like integers, longs, and doubles
Additions, Subtractions	Up to 5X speedup for additions and subtractions
Multiplications	Up to 10X speedup for multiplications
Divisions	Up to 30X speedup for divisions
Scale modification	Up to 2X speedup for the setScale function
Rounding	Up to 4X speedup for rounding

The above speedups are reported from measurements taken using 32- and 64-bit versions of IBM SDK for Java.

Summary

The new IBM POWER6 processor, fully exploited and optimized by IBM's industry-leading SDK for Java 5.0 or 6.0, offers a superior platform for Java applications and middleware. With significant hardware evolution from previous POWER processors, and revolutionary new features such as decimal floating-point, the POWER6 has broken new ground to become the leading hardware platform for Java. This combination offers a seamless upgrade path for Java applications from the POWER4 and POWER5 processors or other platforms, to a new level of performance and reliability with a POWER6 processor-based system.

This paper provides suggested practices for and performance results of using IBM SDK for Java on the POWER6 processor. It discusses memory-related performance and tuning, providing suggestions for tuning the Java heap and descriptions of various garbage collector configurations. It also describes thread-related performance tuning, touching on the AIX threading model used by Java, as well as how dynamic logical partitions are exploited. The paper also provides a description of improvements to startup times, general Java coding guidelines and exploitation of various POWER6 hardware features including simultaneous multi-threading and decimal floating-point hardware.

By employing shared classes and AOT code with Java, we have seen up to 50% improvement in startup time for large middleware applications, and up to 14% for Eclipse startup times. With simultaneous multithreading enabled and large page support, we have seen 6-10% performance improvement with 64 KB page sizes, 8-16% improvement with 16 MB page sizes, and up to 10% improvement with 64 KB pages for compiled Java methods. The processing time of a modified version of jTelco decreased approximately 45% with the POWER6 DFU and other BigDecimal microbenchmarks show significant speedups in the range of 5x-30x for arithmetic operations.

Resources

Memory-Related Performance and Tuning

Tuning the Java Heap

- Tuning Java heap for performance on AIX
http://www16.boulder.ibm.com/pseries/en_US/aixprgpd/genprogdc/genprogctfrm.htm

Garbage Collection

- A Brief Introduction to the IBM Developer Kit for Java 5.0
Available by request from the authors
- Java technology, IBM style: Garbage collection policies, Part 1
<http://www.ibm.com/developerworks/java/library/j-ibmjv2/index.html>
- Java technology, IBM style: Garbage collection policies, Part 2
<http://www.ibm.com/developerworks/java/library/j-ibmjv3/index.html>
- IBM SDK 5 Diagnostics Guide

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

Large Pages

- http://www.ibm.com/servers/aix/whitepapers/large_page.html
- http://www.ibm.com/servers/aix/whitepapers/multiple_page.pdf
- <http://www.ibm.com/developerworks/db2/library/techarticle/dm-0606kamath/>

Thread-Related Tuning

Threading Models

- Processor scheduler performance:
http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prfungd/doc/prfungd/sched_policy_threads.htm
- Thread environment variables:
http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prfungd/doc/prfungd/thread_env_vars.htm
- AIX 5L Performance Tools Handbook (An IBM Redbooks publication):
<http://www.redbooks.ibm.com/redbooks/pdfs/sg246039.pdf>
- The schedo command. Commands Reference, Volume 5:
<http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.cmds/doc/aixcmds5/schedo.htm>
- Diagnostics Guide 6.0:
<http://www.ibm.com/developerworks/java/jdk/diagnosis/>
<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag60.pdf>

Exploiting Dynamic Logical Partitions (Dynamic LPAR) with Java

- Exploiting Dynamic LPAR with Java
<http://www.ibm.com/servers/eserver/pseries/hardware/whitepapers/dlpar.html>
- IBM SDK for Java: docs/apidoc directory

Startup Performance

- IBM SDK Class Sharing
<http://www.ibm.com/developerworks/java/library/j-ibmjava4/>
- IBM SDK User Guide
<http://www.ibm.com/developerworks/java/jdk/docs.html>
- IBM Diagnostics Guide 6.0
<https://publib2.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp>

Java Coding Guidelines

- Java Coding Guidelines
<http://jcp.org/en/jsr/detail?id=133>

POWER6 Hardware Feature Exploitation

Simultaneous Multithreading

- <http://www.ibm.com/servers/eserver/iserries/perfmgmt/pdf/SMT.pdf>
- <http://www.research.ibm.com/journal/rd/494/sinharoy.html>
- <http://www.research.ibm.com/journal/rd/494/mathis.pdf>
- <http://www.itjungle.com/breaking/bn101106-story01.html>

Decimal Floating-Point Unit Exploitation

- IBM Decimal Arithmetic
<http://www.hursley.ibm.com/decimal/>
- A Decimal Floating-Point Specification
<http://www2.hursley.ibm.com/decimal/arith15-foils.pdf>
- IEEE754R Working Group
<http://www.validlab.com/754R/>
- Java 6 BigDecimal API
<http://java.sun.com/javase/6/docs/api/java/math/BigDecimal.html>
- “POWER6 Accelerators: VMX and DFU”
To be published in IBM Journal of Research and Development, Late 2007
- IBM Decimal Floating-Point Resource Center
<http://www.hursley.ibm.com/decimal>
- jTelco
<http://www.hursley.ibm.com/decimal/telco.html>

About the Authors

Venkata Ravi Kumar Dadi is an Advisory Software Engineer for the IBM System p Business Strategy and Technical enablement organization at IBM. He is based in Silicon Valley, Bay Area California. His main role is to help solution developers bring their applications to AIX on POWER. While working at IBM, he has held several positions and he is currently working on enabling Oracle applications on System p software and hardware platforms.

Levon Stepanian is a software developer for the IBM Java just-in-time compiler. Based out of the IBM Software Lab located in Markham, Ontario, he completed his M.Sc. at the University of Toronto, working on Java Native call inlining in the IBM Java just-in-time compiler, and now spends his development time between p and zSeries platforms.



© IBM Corporation 2007
IBM Corporation
Systems and Technology Group
Route 100
Somers, New York 10589

Produced in the United States of America
June 2007
All Rights Reserved

This document was developed for products and/or services offered in the United States. IBM may not offer the products, features, or services discussed in this document in other countries.

The information may be subject to change without notice. Consult your local IBM business contact for information on the products, features and services available in your area.

All statements regarding IBM future directions and intent are subject to change or withdrawal without notice and represent goals and objectives only.

IBM, the IBM logo, AIX, AIX 5L, OS/400, Power Architecture, POWER, POWER4, POWER5, POWER5+, POWER6, S/390, System i, System p, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries or both. A full list of U.S. trademarks owned by IBM may be found at: <http://www.ibm.com/legal/copytrade.shtml>

The Power Architecture and Power.org wordmarks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

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 of Sun Microsystems, Inc. in the United States and/or other countries.

AltiVec is a trademark of Freescale Semiconductor, Inc.

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

Copying or downloading the images contained in this document is expressly prohibited without the written consent of IBM.

Information concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of the non-IBM products should be addressed with those suppliers.

All performance information was determined in a controlled environment. Actual results may vary. Performance information is provided "AS IS" and no warranties or guarantees are expressed or implied by IBM. Buyers should consult other sources of information, including system benchmarks, to evaluate the performance of a system they are considering buying.

When referring to storage capacity, 1 TB equals total GB divided by 1000; accessible capacity may be less.

The IBM home page on the Internet can be found at: <http://www.ibm.com>.

The IBM System p home page on the Internet can be found at: <http://www.ibm.com/systems/p>.

PSW03010-USEN-01