# Java Tuning White Paper

**Java™ Enterprise Platforms and Developer Organization**

**Sun Microsystems, Inc.**

**Revised: December 20, 2005**

## Table of Contents

## 1 Introduction

This Java Tuning White Paper is intended as a reference for Java Performance Tuning information, techniques and pointers.

### 1.1 Goals

The goals of this White Paper are to collect the best practices and "How To" for Java Performance in one place.

The initial target for this tuning document is tuning server applications on large, multi-processor servers. Future versions of this document will explore similar recommendations for desktop Java performance.

### 1.2 This is a Living Document

This White Paper has been written with the latest advances in Sun's Java™ HotSpot™ Virtual Machine in mind. Therefore this document will evolve on a frequent basis to reflect the latest performance features and new best practices.

### 1.3 How to Use this White Paper

This White Paper is organized in sections from the easiest, most accessible ways of getting better Java performance to progressively more complex tuning and design recommendations.

Start with Best Practices to insure you are getting the best Java performance possible even before you do any tuning. Before you dive into performance tuning it's essential to understand the right ways of Making Decisions from Data. Only with that basis in directed analysis can you safely proceed to explore Tuning Ideas for your application. Why are there "Tuning Ideas" and not just blanket recommendations? Because every application is different and no one set of recommendations is right for every deployment environment. The Tuning Ideas section is intended to give you not just Java command line options but also background on what they mean and when they may lead to increased performance.

Even during the tuning process, but certainly as you want to take performance to the next level it will be necessary to explore Monitoring and Profiling your application. By gathering detailed data on actual application performance you can fine tune command line options and have an idea where to focus coding improvement efforts. In the section on Coding for Performance we will cover Java™ API's to consider when architecting applications with performance in mind.

Many other documents and sites will be collected in the Pointers section. We encourage you to help continue making Java faster through Feedback and the Java Performance Community. Your feedback will help make this White Paper better.

# 2 Best Practices

There are many best practices for getting the optimal performance with your Java application. Here are some very basic practices that can make a significant difference in performance before you begin tuning.

### 2.1 Use the most recent Java™ release

Each major release of Java™ introduces new features, bug fixes and performance enhancements. Before you begin tuning or consider application level changes to take advantage of new language features it is still worthwhile to upgrade to the latest major Java™ release. For information on why upgrading to Java SE 5.0 (Tiger) is important please see the references in the Pointers section.

Understandably this is not always possible as certain applications, especially third party ISV applications may not yet provide support for the latest release of Java™. In that case use the most recent *update* release which is supported for your application (and please encourage your ISV to support the latest Java™ version!).

### 2.2 Get the latest Java™ update release

For each major Java™ release "train" (e.g. J2SE 1.3.1, J2SE 1.4.2, J2SE 5.0) Sun publishes update releases on a regular basis. For example the most recent update release of J2SE 5.0 is *update 6* or Java SE 1.5.0_06.

Update releases often include bug fixes and performance improvements. By deploying the most recent update release for Java™ you will benefit from the latest and greatest performance improvements.

### 2.3 Insure your operating system patches are up-to-date

Even though Java is cross platform it does rely on the underlying operating system and therefore it is important that the OS basis for the Java™ Platform is as up-to-date as possible.

In the case of Solaris there is a set of patches that are recommended when deploying Java applications. To get these Solaris patches for Java please see the links under the section Solaris OS Patches on the Java™ download page.

### 2.4 Eliminate variability

Be aware that various system activities and the operations of other applications running on your system can introduce significant variance into the measurements of any application's performance, including Java applications. The activities of the OS and other applications may introduce CPU, Memory, disk or network resource contention that may interfere with your measurements.

Before you begin to measure Java performance try to assess application behavior that may *discolor* your performance results (e.g. accessing the Internet for updates, reading files from the users home directory, etc.). By simplifying application behavior as much as possible and by changing only one variable at a time (i.e. operating system tunable parameter, Java command line option, application argument, etc.) your performance investigation can track the impact of each change independently.

# 3 Making Decisions from Data

It is really tempting to run an application once before and once after a change and draw some conclusion about the impact of that change. Sometimes the application runs for a long time. Sometimes launching the application may be complex and dependent on multiple external services. But can you legitimately make a decision from this test? It's really impossible to know if you can safely draw a conclusion from the data unless you measure the power of the data quantitatively. Applying the scientific method is important when designing any set of experiments. Rigor is especially necessary when measuring Java application performance because the behavior of Java™ HotSpot™ virtual machine adapts and reacts to the specific machine and specific application it is running. And subtle changes in timing due to other system activity can result in measurable differences in performance and which are unrelated to the comparisons being made.

### 3.1 Beware of Microbenchmarks!

One of the advantages of Java is that it dynamically optimizes for data at runtime. More and more we are finding cases where Java performance meets or exceeds the performance of similar statically compiled programs. However this adaptability of the JVM makes it hard to measure small snippets of Java functionality.

One of the reasons that it's challenging to measure Java performance is that it changes over time. At startup, the JVM typically spends some time "warming up". Depending on the JVM implementation, it may spend some time in interpreted mode while it is profiled to find the 'hot' methods. When a method gets sufficiently hot, it may be compiled and optimized into native code.

Indeed some of these optimizations may unroll loops, hoist variables outside of loops or even eliminate "dead code" altogether. So how should you handle this? Should you factor in the differences in machine speed by timing a few iterations of loop and then run a number of iterations that will give you similar total run time on each platform? If you do that what is likely to happen is your timing loop will estimate loop time while running in interpreted mode. Then when running the test the loop will get optimized and run much more quickly. So quickly, in fact, that your total run time might be so short that you are not measuring the inner loop at all but just the infrastructure for warming up the application.

For certain applications garbage collection can complicate writing benchmarks. It is important to note, however, that for a given set of tuning parameters that GC throughput is predictable. So either avoid object allocations in your inner loop (to avoid provoking GC's) or run long enough to reach GC steady state. If you do allocate objects as part of the benchmark be careful to size the heap as to minimize the impact of GC and gather enough samples so that you get a fair average for how much time is spent in GC.

There are even more subtle traps with benchmarking. What if the work inside the loop is not really constant for each iteration? If you append to a string, for example, you may be doing a copy before append which will increase the amount of work each time the loop is executed. Remember to try to make the computation in the loop constant and non-trivial. By printing the final answer you will prevent the entire loop body from being completely eliminated.

Clearly running to steady state is essential to getting repeatable results. Consider running the benchmark for several minutes. Any application which runs for less than one minute is likely to be dominated by JVM startup time. Another timing problem that can occur is jitter. Jitter occurs when the granularity of the timing mechanism is much coarser than the variability in benchmark timing. On certain Windows platforms, for example, the call `System.currentTimeMillis()` has an effective 15 ms granularity. For short tests this will tend to discolor results. Any new benchmarks should take advantage of the new `System.nanoTime()` method which should reduce this kind of jitter.

### 3.2 Use Statistics

Prior to the experiment you should try to eliminate as much application performance variability as possible. However it is rarely possible to eliminate *all* variability, especially noise from asynchronous operating system services. By repeating the same experiment over the course of several trials and averaging the results you effectively focus on the signal instead of the noise. The rate of improving the signal is proportional to the square root of the number of samples (for more on this see Reducing the Effects of Noise in a Data Acquisition System by Averaging ).

To put Java benchmarking into statistics terms we are going to test the *baseline* settings before a change and the *specimen* settings after a change. For example you may run a benchmark baseline test 10 times with no Java command line options and specimen test 10 times with the command line of "-server". This will give you two different sample populations. The questions you really want to answer are, "Did that change in Java settings make a difference? And, if so, how much of a difference?".

The second question, determining the percentage improvement is actually the easier question:
`percentageImprovement = 100.0 x (SpecimenAvg - BaselineAvg) / BaselineAvg`

The first question, "Did that change in Java settings make a difference?", is the most important question, however because what we really want to know is "Is the difference significant enough that we can safely draw conclusions from it?". In statistics jargon this question could be rephrased as "Do these two sample populations reflect the same underlying population or not?". To answer this question we use the Students t-test. See the Pointers section for more background on the Student's t-test. Using the number of samples, mean value and standard deviation of the baseline population and the specimen population as well as setting the desired risk level ( *alpha*) we can determine the *p-value* or probability that the change in Java settings was significant. The risk level ( *alpha*) is often set at 0.05 (or 0.01) which means that five times (one time) out of one hundred you would find a statistically significant difference between the means even if there was none. Generally speaking if the *p-value* is less than 0.05 then we would say that the difference is significant and thus we can draw the conclusion that the change in Java settings did make a difference. For more on interpreting p-values and power analysis please see The Logic of Hypothesis Testing.

### 3.3 Use a benchmark harness

What is a *benchmark harness*? A benchmark harness is often a script program that you use to launch a given benchmark, capture the output log files, and extract the benchmark score (and sub-scores). Often a benchmark harness will have the facility to run a benchmark a predetermined number of trials and, ideally, calculate statistics based the results of different Java settings (whether the settings changes are at the OS level, Java tuning or coding level differences).

Some benchmarks have a harness included. Even in those cases you want to re-write the harness to test a wider variety of parameters, capture additional data and/or calculate additional statistics. The advantage of writing even a simple benchmark harness is that it can remove the tedium of gathering many samples, it can launch the application in a consistent way and it can simplify the process of calculating statistics.

Whether you use a benchmark harness or not it will be essential to insure that when you attempt Java tuning changes that you are able to answer the question: "Have I gathered enough samples to provide enough statistical significance that I can draw a conclusion from the result?".

# 4 Tuning Ideas

By now you have taken the easy steps in the Best Practices section and have prepared for tuning by understanding the right ways of Making Decisions from Data. This section on Tuning Ideas contains suggestions on various tuning options that you should try on with your Java application. All comparisons made between different sets of options should be performed using the statistical techniques discussed above.

### 4.1 General Tuning Guidelines

Here are some general tuning guidelines to help you categorize the kinds of Java tuning you will perform.

#### 4.1.1 Be Aware of Ergonomics Settings

Before you start to tune the command line arguments for Java be aware that Sun's HotSpot™ Java Virtual Machine has incorporated technology to begin to tune itself. This smart tuning is referred to as Ergonomics. Most computers that have at least 2 CPU's and at least 2 GB of physical memory are considered *server-class* machines which means that by default the settings are:
The `-server` compiler
The `-XX:+UseParallelGC` parallel (throughput) garbage collector
The `-Xms` initial heap size is 1/64th of the machine's physical memory
The `-Xmx` maximum heap size is 1/4th of the machine's physical memory (up to 1 GB max).
Please note that 32-bit Windows systems all use the `-client` compiler by default and 64-bit Windows systems which meet the criteria above will be be treated as server-class machines.

#### 4.1.2 Heap Sizing

Even though Ergonomics significantly improves the "out of the box" experience for many applications, optimal tuning often requires more attention to the sizing of the Java memory regions.

The maximum heap size of a Java application is limited by three factors: the process data model (32-bit or 64-bit) and the associated operating system limitations, the amount of virtual memory available on the system, and the amount of physical memory available on the system. The size of the Java heap for a particular application can never exceed or even reach the maximum virtual address space of the process data model. For a 32-bit process model, the maximum virtual address size of the process is typically 4 GB, though some operating systems limit this to 2 GB or 3 GB. The maximum heap size is typically -Xmx3800m (1600m) for 2 GB limits), though the actual limitation is application dependent. For 64-bit process models, the maximum is essentially unlimited. For a single Java application on a dedicated system, the size of the Java heap should never be set to the amount of physical RAM on the system, as additional RAM is needed for the operating system, other system processes, and even for other JVM operations. Committing too much of a system's physical memory is likely to result in paging of virtual memory to disk, quite likely during garbage collection operations, leading to significant performance issues. On systems with multiple Java processes, or multiple processes in general, the sum of the Java heaps for those processes should also not exceed the the size of the physical RAM in the system.

The next most important Java memory tunable is the size of if the young generation (also known as the NewSize). Generally speaking the largest recommended value for the young generation is 3/8 of the maximum heap size. Note that with the throughput and low pause time collectors it may be possible to exceed this ratio. For more information please see the discussions of the Young Generation Guarantee in the Tuning Garbage Collection with the 5.0 Java Virtual Machine document.

Additional memory settings, such as the stack size, will be covered in greater detail below.

#### 4.1.3 Garbage Collector Policy

The Java™ Platform offers a choice of Garbage Collection algorithms. For each of these algorithms there are various policy tunables. Instead of repeating the details of the Tuning Garbage Collection document here suffice it to say that first two choices are most common for large server applications:
The `-XX:+UseParallelGC` parallel (throughput) garbage collector, or
The `-XX:+UseConcMarkSweepGC` concurrent (low pause time) garbage collector (also known as CMS)
The `-XX:+UseSerialGC` serial garbage collector (for smaller applications and systems)

#### 4.1.4 Other Tuning Parameters

Certain other Java tuning parameters that have a high impact on performance will be mentioned here. Please see the Pointers section for a comprehensive reference of Java tuning parameters.

The VM Options page discusses Java support for Large Memory Pages. By appropriately configuring the operating system and then using the command line options `-XX:+UseLargePages` ( *on by default for Solaris*) and `-XX:LargePageSizeInBytes` you can get the best efficiency out of the memory management system of your server. Note that with larger page sizes we can make better use of virtual memory hardware resources (TLBs), but that may cause larger space sizes for the Permanent Generation and the Code Cache, which in turn can force you to reduce the size of your Java heap. This is a small concern with 2 MB or 4 MB page sizes but a more interesting concern with 256 MB page sizes.

An example of a Solaris-specific tunable is selecting the **libumem** alternative heap allocator. To experiment with **libumem** on Solaris you can use the following LD_PRELOAD environment variable directive:
To set **libumem** for all child processes of a given shell, set and export the environment variable
`LD_PRELOAD=/usr/lib/libumem.so`
To launch a Java application with **libumem** from **sh**:
`LD_PRELOAD=/usr/lib/libumem.so java java-settings application-args`
To launch a Java application with **libumem** from **csh**:
`env LD_PRELOAD=/usr/lib/libumem.so java java-settings application-args`
You can verify that **libumem** is in use by verifying the process settings with **pmap(1)** or **pldd(1)**.

## 4.2 Tuning Examples

Here are some specific tuning examples for your experimentation. Please understand that these are only examples and that the optimal heap sizes and tuning parameters for your application on your hardware may differ.

### 4.2.1 Tuning Example 1: Tuning for Throughput

Here is an example of specific command line tuning for a server application running on system with 4 GB of memory and capable of running 32 threads simultaneously (CPU's and cores or contexts).

```
java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20
```

Comments:
**`-Xmx3800m -Xms3800m`**
Configures a large Java heap to take advantage of the large memory system.
**`-Xmn2g`**
Configures a large heap for the young generation (which can be collected in parallel), again taking advantage of the large memory system. It helps prevent short lived objects from being prematurely promoted to the old generation, where garbage collection is more expensive.
**`-Xss128k`**
Reduces the default maximum thread stack size, which allows more of the process' virtual memory address space to be used by the Java heap.
**`-XX:+UseParallelGC`**
Selects the parallel garbage collector for the new generation of the Java heap (note: this is generally the default on server-class machines)
**`-XX:ParallelGCThreads=20`**
Reduces the number of garbage collection threads. The default would be equal to the processor count, which would probably be unnecessarily high on a 32 thread capable system.

### 4.2.2 Tuning Example 2: Try the parallel old generation collector

Similar to example 1 we here want to test the impact of the parallel old generation collector.

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -
XX:+UseParallelOldGC
```

Comments:
**`-Xmx3550m -Xms3550m`**
Sizes have been reduced. The ParallelOldGC collector has additional native, non-Java heap memory requirements and so the Java heap sizes may need to be reduced when running a 32-bit JVM.
**`-XX:+UseParallelOldGC`**
Use the parallel old generation collector. Certain phases of an old generation collection can be performed in parallel, speeding up a old generation collection.

### 4.2.3 Tuning Example 3: Try 256 MB pages

This tuning example is specific to those Solaris-based systems that would support the huge page size of 256 MB.

```
java -Xmx2506m -Xms2506m -Xmn1536m -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -
XX:+UseParallelOldGC -XX:LargePageSizeInBytes=256m
```

Comments:
**`-Xmx2506m -Xms2506m`**
Sizes have been reduced because using the large page setting causes the permanent generation and code caches sizes to be 256 MB and this reduces memory available for the Java heap.
**`-Xmn1536m`**
The young generation heap is often sized as a fraction of the overall Java heap size. Typically we suggest you start tuning with a young generation size of 1/4th the overall heap size. The young generation was reduced in this case to maintain a similar ratio between young generation and old generation sizing used in the previous example option used.
**`-XX:LargePageSizeInBytes=256m`**
Causes the Java heap, including the permanent generation, and the compiled code cache to use as a minimum size one 256 MB page (for those platforms which support it).

### 4.2.4 Tuning Example 4: Try -XX:+AggressiveOpts

This tuning example is similar to Example 2, but adds the AggressiveOpts option.

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -
XX:+UseParallelOldGC -XX:+AggressiveOpts
```

Comments:
**`-Xmx3550m -Xms3550m`**
Sizes have been increased back to the level of Example 2 since we no longer using huge pages.
**`-Xmn2g`**
Sizes have been increased back to the level of Example 2 since we no longer using huge pages.
**`-XX:+AggressiveOpts`**
Turns on point performance optimizations that are expected to be on by default in upcoming releases. The changes grouped by this flag are minor changes to JVM runtime compiled code and not distinct performance features (such as BiasedLocking and ParallelOldGC). This is a good flag to try the JVM engineering team's latest performance tweaks for upcoming releases. Note: this option is *experimental*! The specific optimizations enabled by this option can change from release to release and even build to build. You should reevaluate the effects of this option with prior to deploying a new release of Java.

### 4.2.5 Tuning Example 5: Try Biased Locking

This tuning example is builds on Example 4, and adds the Biased Locking option.

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -
XX:+UseParallelOldGC -XX:+AggressiveOpts -XX:+UseBiasedLocking
```

Comments:
**`-XX:+UseBiasedLocking`**
Enables a technique for improving the performance of uncontended synchronization. An object is "biased" toward the thread which first acquires its monitor via a **monitorenter** bytecode or synchronized method invocation; subsequent monitor-related operations performed by that thread are relatively much faster on multiprocessor machines. Some applications with significant amounts of uncontended synchronization may attain significant speedups with this flag enabled; some applications with certain patterns of locking may see slowdowns, though attempts have been made to minimize the negative impact.

#### 4.2.6 Tuning Example 6: Tuning for low pause times and high throughput

This tuning example similar to Example 2, but uses the concurrent garbage collector (instead of the parallel throughput collector).

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -
XX:+UseParNewGC -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=31
```

Comments:
**`-XX:+UseConcMarkSweepGC -XX:+UseParNewGC`**
Selects the Concurrent Mark Sweep collector. This collector may deliver better response time properties for the application (i.e., low application pause time). It is a parallel and mostly-concurrent collector and and can be a good match for the threading ability of an large multi-processor systems.
**`-XX:SurvivorRatio=8`**
Sets survivor space ratio to 1:8, resulting in larger survivor spaces (the smaller the ratio, the larger the space). Larger survivor spaces allow short lived objects a longer time period to die in the young generation.
**`-XX:TargetSurvivorRatio=90`**
Allows 90% of the survivor spaces to be occupied instead of the default 50%, allowing better utilization of the survivor space memory.
**`-XX:MaxTenuringThreshold=31`**
Allows short lived objects a longer time period to die in the young generation (and hence, avoid promotion). A consequence of this setting is that minor GC times can increase due to additional objects to copy. This value and survivor space sizes may need to be adjusted so as to balance overheads of copying between survivor spaces versus tenuring objects that are going to live for a long time. The default settings for CMS are **SurvivorRatio=1024** and **MaxTenuringThreshold=0** which cause all survivors of a scavenge to be promoted. This can place a lot of pressure on the single concurrent thread collecting the tenured generation. Note: when used with `-XX:+UseBiasedLocking`, this setting should be 15.

#### 4.2.7 Tuning Example 7: Try AggressiveOpts for low pause times and high throughput

This tuning example is builds on Example 6, and adds the AggressiveOpts option.

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -
XX:+UseParNewGC -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=31 -
XX:+AggressiveOpts
```

Comments:
**`-XX:+AggressiveOpts`**
Turns on point performance optimizations that are expected to be on by default in upcoming releases. The changes grouped by this flag are minor changes to JVM runtime compiled code and not distinct performance features (such as BiasedLocking and ParallelOldGC). This is a good flag to try the JVM engineering team's latest performance tweaks for upcoming releases. Note: this option is *experimental*! The specific optimizations enabled by this option can change from release to release and even build to build. You should reevaluate the effects of this option with prior to deploying a new release of Java.

## 5 Monitoring and Profiling

Discussing monitoring (extracting high level statistics from a running application) or profiling (instrumenting an application to provide detailed performance statistics) are subjects which are worthy of White Papers in their own right. For the purpose of this Java Tuning White Paper these subjects will be introduced using tools as examples which can be used on a permanent basis without charge.

### 5.1 Monitoring

The Java™ Platform comes with a great deal of monitoring facilities built-in. Please see the document Monitoring and Management for the Java™ Platform for more information.

The most popular of these "built-in" tools are JConsole and the jvmstat technologies.

### 5.2 Profiling

The Java™ Platform also includes some profiling facilities. The most popular of these "built-in" profiling tools are The `-Xprof` Profiler and the HPROF profiler (for use with HPROF see also Heap Analysis Tool).

A profiler based on JFluid Technology has been incorporated into the popular NetBeans development tool.

## 6 Coding for Performance

This section will cover coding level changes that you can make which will make an impact on performance. For the purpose of this initial draft of the Java Tuning White Paper examples of the kinds of coding level changes that can have an impact on performance are taking advantage of new language features like NIO and the Concurrency utilities.

The New I/O API's (or NIO) offer improved performance for operations like memory mapped files and scalable network operations. By using NIO developers may be able to significantly improve performance of memory or network intensive applications. One of the success stories of using NIO is in the Grizzly web container which is part of Sun's GlassFish project.

Another example new Java language features that impact performance is the set of Concurrency Utilities. Increasingly server applications are going to be targeting platforms with multiple CPU's and multiple cores per CPU. In order to best take advantage of these systems applications must be designed with multi-threading in mind. Classical multi-threaded programming is very complex and error prone due to subtleties in thread interactions such as race conditions. Now with the Concurrency Utilities developers finally have a solid set of building blocks upon which to build scalable multi-threaded applications while avoiding much of the complexity of writing a multi-threaded framework.

## 7 Pointers

Here are the resources that have been referenced in this document.
Getting and Deploying the latest version of Java

Download the latest version of Java
Java Deployment for J2SE 5.0
Java Web Start for J2SE 5.0
Java Plug-in for J2SE 5.0
The J2SE 5.0 Platform
J2SE 5.0 Adoption
Reasons to Migrate to J2SE 5.0 (Tiger)
J2SE 5.0 Supported System Configurations
J2SE Platform Overview
J2SE 5.0 Overview
J2SE 5.0 Documentation Guide
Java Naming Conventions
Understanding Statistics
Wikipedia: Student's t-test
Statistics Every Writer Should Know including Student's T
HyperStat Online: An Introductory Statistics Textbook
Java Tuning and Ergonomics
Ergonomics in the 5.0 Java Virtual Machine
Server-Class Machine Detection
Garbage Collector Ergonomics
Standard Java Options
Java HotSpot VM Options
Joe Mocker's Collection of JVM Options
Garbage Collection
Tuning Garbage Collection with the 5.0 Java Virtual Machine
Monitoring, Management and Profiling
Monitoring and Management for the Java™ Platform
Using JConsole to Monitor Applications
jvmstat 3.0 and visualgc - Visual Garbage Collection Monitoring Tool
The `-Xprof` Profiler
The HPROF profiler and Heap Analysis Tool
The NetBeans Profiler
Synthetic Load Generation
SLAMD Distributed Load Generation Engine
Coding for Performance
New Features and Enhancements J2SE 5.0
Java 5.0 Concurrency Utilities
Java Performance Documentation
Java Performance Tech Topic
J2SE 5.0 Performance White Paper
Java Performance Community

## 8 Feedback and the Java Performance Community

As this White Paper is a living document we hope to continually improve it by adding additional Best Practices and Tuning Ideas as well as additional Pointers.

### 8.1 Feedback

Do you have a comment, question, bug or enhancement for this White Paper? Please submit your comment in the General Performance Discussion forum on java.net and use "Java Tuning White Paper" in the subject. Following discussions of the initial draft we have started a Wiki page on java.net to discuss corrections to this white paper and ideas for future versions: PerformanceTuningWhitePaper

### 8.2 Java Performance Community

Do you have performance questions that are not answered by this document? Would like to contribute code that helps runs Java applications (a harness) and/or calculates statistics? Please post your question or idea in the General Performance Discussion forum. The forum is the place where the Java Performance Community can share ideas and best practices.

E-mail this page Printer View