

WHITE PAPER

# Java in Virtual Machines on VMware® ESX: Best Practices



## TABLE OF CONTENTS

<b>1. SUMMARY OF BEST PRACTICES.....</b>	<b>1</b>
1.1 Java in Virtual Machines on ESX .....	1
1.2. Running Applications in ESX Virtual Machines .....	2
<b>2. INTRODUCTION .....</b>	<b>2</b>
<b>3. JAVA AND SYSTEMS ARCHITECTURE .....</b>	<b>3</b>
3.1. Multi-Tiered Applications .....	3
3.2. Instances of Java in Virtual Machines .....	4
<b>4. MEMORY .....</b>	<b>4</b>
4.1. Java Heap Memory.....	5
4.2. Virtual Machine Memory.....	6
4.2.1. Sizing Virtual Machine Memory.....	6
4.2.2. Memory Reservation.....	6
4.2.3. Large Memory Pages .....	7
4.3. ESX Memory.....	8
4.3.1. Memory Overhead .....	8
4.4. Examining Memory Consumption .....	9
<b>5. VIRTUAL CPUs AND THREADS .....</b>	<b>10</b>
5.1. Matching Threads and Virtual CPUs .....	10
5.2. Virtual CPU Recommendations .....	11
<b>6. DISK I/O.....</b>	<b>11</b>
6.1. Disk I/O Recommendation .....	11
<b>7. TIMEKEEPING .....</b>	<b>11</b>
7.1. Timekeeping Recommendations.....	11
7.2. Effects of Timer interrupts .....	12
7.3. The /usepmtimer option with Windows Guest Operating Systems .....	13
<b>8. MICRO-BENCHMARKS .....</b>	<b>14</b>
8.1. Warm-up Time .....	14
<b>9. CONCLUSIONS .....</b>	<b>14</b>
<b>10. REFERENCES.....</b>	<b>15</b>
<b>11. APPENDIX 1 : CHECKLIST FOR ISSUE RESOLUTION .....</b>	<b>16</b>

# 1. Summary of Best Practices

This paper discusses best practices for running Java-based software in VMware® ESX virtual machines. These guidelines will help you to get the best from your Java applications and application servers when you run them on VMware® Infrastructure 3. Java applications have been found to perform very well in virtual machines on ESX, coming close to native performance in many cases. The **main difference** between running Java applications in virtual machines on ESX and running those same applications on physical systems essentially boils down to

- the choice of memory size for the ESX virtual machine and
- the number of virtual CPUs used in the ESX virtual machine.

All the best practices that you use when running Java on physical systems apply equally in the virtual machine case. We assume that you have tuned your Java application to run well on the physical system from which you are redeploying it to a virtual machine (refs 1, 2 and 3). The body of this paper provides the technical background and justification for the best practices recommended here.

## 1.1 Java in Virtual Machines on ESX

This section summarizes the practices that apply specifically for Java applications in an ESX virtual machine.

### Memory

- Size the virtual machine memory to leave adequate space for
  - the Java heap;
  - the other memory demands of the Java Virtual Machine code and thread stacks;
  - any other concurrently executing process that needs memory;
  - the guest operating system.
- Set the memory “Reservation” value using the VMware® Infrastructure Client to the size of memory for the virtual machine.
- Use large memory pages by informing the Java virtual machine (JVM) and guest operating system that they are being used. ESX 3.5 supplies large pages by default where possible when the guest operating system requests them.

### Virtual CPUs

- Determine the optimum number of **virtual** CPUs for a virtual machine that hosts a Java application by testing the virtual machine configured with different numbers of virtual CPUs with the same test load.
- If you are using multiple garbage collector (GC) threads in your JVM (such as those occasions when you use a parallel garbage collector), then the number of garbage collector threads should be equal to or less than the number of virtual CPUs that are configured in the virtual machine.

### Management

- For easier **monitoring** and load balancing, use one JVM process per ESX virtual machine.
- Use the lower resolution timing Java options supplied by your JVM.

## 1.2. Running Applications in ESX Virtual Machines

The recommendations given in this section apply to all applications that are run in virtual machines on ESX. Java applications are one important example of these.

### Memory

- Use VMware® Distributed Resource Scheduler to balance the virtual machines' memory requirements across the cluster.

### Virtual CPUs

- Use the lowest number of virtual CPUs that is practical for your application.

### Disk I/O

- Check the guest average (GAVG) and disk average (DAVG) latency time columns in the **esxtop** tool's output to ensure that your I/O system is not causing bottlenecks due to disk latencies.

### Timekeeping

- Synchronize the time on the ESX host with an external NTP source.
- Synchronize the time in the virtual machine with an external NTP source for Linux guests and using `w32time` for Windows guest operating systems.
- Use a lower clock interrupt rate in the guest operating system within the virtual machine.
- Avoid using the `/usepmtimer` option in the `boot.ini` system configuration for Windows guest operating systems that use an SMP HAL

## 2. Introduction

As a developer or deployer of systems, you can move your Java applications from physical systems and run them very easily on VMware ESX, the virtualization layer of VMware Infrastructure 3. The experiences of customers who have already virtualized their Java-based systems in large numbers confirms that virtualized Java applications behave the same way they do on operating systems that run on physical hardware (or "physical").

You can re-create your current physical Java environment on one or more virtual machines hosted on VMware ESX without making any changes to the application code. Java applications perform well on ESX virtual machines without adjusting the arguments that you use when they run on physical operating systems. To help you get started with redeploying Java applications on to virtual machines, this document:

- gathers together a set of best practices based on experience deploying Java systems on virtual machines.
- offers health checks and runtime options that you can use on your Java-based application to ensure that it runs well on virtual platforms.
- makes recommendations that help you deploy Java on virtual machines.

The best practices described here require no changes to the application code.

Java-based software is being run on VMware Infrastructure in all kinds of industries, from health care to insurance, banking and gaming. Companies have deployed into production dozens or even hundreds of ESX virtual machines with Java as the main application platform within them.

Computing resources such as CPU, memory and I/O devices are configured differently in virtual machines than they are in physical systems. There may be different numbers of virtual CPUs in a virtual machine than there are physical CPUs on the host system, for example – and the number of virtual CPUs or the virtual machine memory size can be adjusted much more rapidly.

In order to get an accurate result when comparing the performance of your Java application on physical and virtual machine implementations, it is important to assign the same number of virtual CPUs to the ESX virtual machine as there are physical CPUs on the physical operating system being tested. It is also necessary to assign the same amount of memory to the ESX virtual machine as was used in the physical tests – otherwise the comparison is not a true one. These and other considerations, described below, are important to bear in mind when moving your Java application to a virtual environment.

Section 3 following contains background information on virtualizing Java applications. Subsequent sections examine each computing resource in turn, starting with memory, followed by threads and virtual CPUs, disk I/O, timekeeping and end with some advice on benchmarking. These sections explain the differences in the use of these resources in their virtualized form and include the recommended best practices. At the end of the document you will find references and a checklist of items to look at when resolving issues with Java on virtual machines on ESX.

## 3. Java and Systems Architecture

Java application performance tuning information is already covered in several books and papers, as well as at the JVM suppliers' websites. The references section at the end of this document provides some sources for Java performance tuning techniques. The best practices for achieving higher performance with Java applications on a physical system are also applicable to a virtual implementation of a Java program. Since it is readily available elsewhere, this performance tuning information is not repeated here.

### 3.1. Multi-Tiered Applications

Java applications and J2EE application servers often form one component of a multi-tiered application. Such applications have several parts and dependencies among those parts. These applications may depend on a web server tier, an application server tier, a database tier and other tiers, such as a load balancer tier. System performance is often under scrutiny both in physical and virtual implementations that depend on Java and Java is sometimes seen as the culprit for any performance issues. Similarly, the virtualization platform is sometimes blamed for performance concerns, although in many cases it is not the source of the issue.

The response times at the database tier often have a determining influence on the speed with which the J2EE application server can send the results of a query back to the end user. On some occasions, the bottleneck in performance is in the database access code (JDBC connection pools) or in the SQL logic that is accessing the data - and not in the Java application logic. At other times, problems with the networking between tiers or the disk I/O configuration can cause a slower response time or lower throughput. It is a good practice to check all parts of the system first before delving into the Java tier for the sources of problems. A systems profiling tool to help with characterizing the parts of the system is invaluable.

The deployment or release engineering teams are often tasked with finding any bottlenecks in performance, which is not always easy to do, since the team may not have been involved in

the architecture of the system or in the coding or testing phases. An application profiling tool (such as VMware® AppSpeed) can help the engineers to look at the various latencies in the whole system first, before delving into one part specifically. A profiling tool will identify where the majority of the time is being spent, for any set of user transactions.

### 3.2. Instances of Java in Virtual Machines

In physical implementations it is not uncommon to see multiple instances of a Java process running concurrently on the same operating system on one machine. This setup may be replicated across many physical machines for scale-out reasons. Java is often the highest consumer of CPU and memory resources on the machine -- it often appears at the top of the list in the "top" output on Linux or in the Task Manager tool on Windows-based systems as the top CPU user.

Installing VMware ESX on a physical machine makes it capable of running a number of virtual machines, all of which are independent of each other from a performance monitoring perspective. In the virtualized world, it is more common to run one instance of Java in any one virtual machine. Nothing prevents you from running two or more instances of Java within a guest operating system in a virtual machine; however, by separating Java instances on individual virtual machines you gain more isolation of potential errors and performance demands, so that you can more readily see the effect of the Java application on the whole system. The equivalent arrangement to having multiple JVMs running on a physical system is to place multiple virtual machines on an ESX host server, each with one JVM in it.

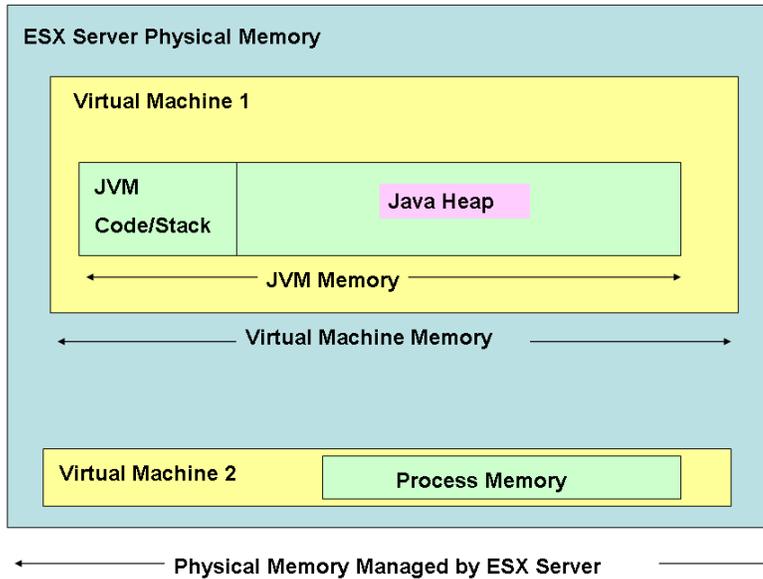
Additionally, the practice of using one JVM per ESX virtual machine also allows VMware® Distributed Resource Scheduler (DRS) to make more intelligent decisions about balancing Java workloads on the ESX host servers in a DRS cluster, since it looks at the resource consumption of the virtual machines to make those decisions.

## 4. Memory

Java programs tend to be very intensive in their use of memory. This has implications for virtualizing your Java process such that you will want to pay attention to

- the Java heap memory size,
- the virtual machine's memory size (in which the Java process will run on the guest operating system) and
- the overall ESX host server's physical memory size.

Figure 1 provides a simplified depiction of the memory layout on an ESX host server running Java within a virtual machine. The host server's physical memory contains all the virtual machines with their own memory, one of which in turn contains the JVM with its memory segments.



**Figure 1: Outline memory layouts with Java running in a virtual machine on an ESX host server**

At the outer level, physical memory is managed by the ESX host. This level represents all of the physical memory that is available on that server, containing the ESX executable code and everything that it manages in the virtual machines. Within that ESX-managed memory there are two virtual machines in this example, one of which is running a Java program. In the virtual machine at the top of the diagram, Java is allocated two different portions of memory, a code/stack section and a heap memory section. JVMs manage most of their objects in the heap memory space. There are other areas of memory in the JVM process, such as the “Permanent” space; these have been omitted here for the sake of clarity. The memory allocations for the virtual machine itself and for the Java heap are determined by certain parameters as described below.

#### 4.1. Java Heap Memory

A certain amount of heap memory is allocated to the JVM at startup time and that memory is accessed throughout the lifetime of the Java program. The starting size for the Java heap is determined by the **-Xms** value and the maximum heap size is set to the **-Xmx** value, both supplied to the JVM at invocation time.

There are different styles of organizing the JVM heap. Memory management algorithms vary across the main Java vendors’ products, and even within one vendor’s JVM. Those objects that no longer have references to them are removed from the heap on a regular basis, as the garbage collector operates. Objects are moved around inside the heap in order to compact the used space, for example, and the free and used heap spaces are adjusted and checked at intervals – whenever the garbage collector runs within the JVM. For these reasons, it appears to the memory management system within the guest operating system in the virtual machine that random accesses are being made to memory addresses. In summary, the JVM needs its entire heap space to be available to it in memory for all of the time it is executing.

As part of your application tuning on physical systems, you will have sized the JVM heap to minimize the number of occurrences and the durations of garbage collection (GC) events, so that both throughput and response times will be optimal. By profiling the runtime behavior of

your Java program under load conditions, you can determine what the working set size of objects is. There are also tools that provide this information in graphical form, such as the Java Monitor tool included with JDK 1.5 onwards.

## 4.2. Virtual Machine Memory

The JVM heap is one component of the overall Java process memory consumption. There is also memory outside the JVM heap space that is occupied by the executable code for the JVM and for the stack of Java methods that are active at any one time. These memory sizes are often much smaller than the JVM heap memory, but they must be considered too.

The extra memory in the virtual machine, beyond the Java heap, is also used for the guest operating system and for other processes that run in that virtual machine.

### 4.2.1. Sizing Virtual Machine Memory

Size the virtual machine memory to leave adequate space for the Java heap, the other memory demands of the JVM code and thread stacks, along with any other concurrently executing process that needs memory from the guest operating system. You will size the ESX virtual machine's memory when you first create it using the VMware Infrastructure® (VI) Client tool. The memory size of the virtual machine can be changed at a later time if necessary. One method for sizing the virtual machine's memory is to take the sum of the following:

1. Space for the guest operating system (This can be 512Mb for a Linux operating system or 1Gb for a Windows OS, for example)
2. Space for the JVM maximum heap size (i.e., the -Xmx value supplied to the JVM)
3. Space for the maximum number of Java threads multiplied by the thread stack size
4. Additional memory for any other programs that are running in the same guest operating system

Default values for the thread stack size can be found by consulting your JVM documentation. The default values depend on whether the JVM is 32-bit or 64-bit. If the application development team wishes, they can override the defaults and specify the thread stack size using the following JVM options:

-Xss<value> (Sun Hotspot JVM and JRockit JVM)  
-Xiss<value> (IBM JVM)

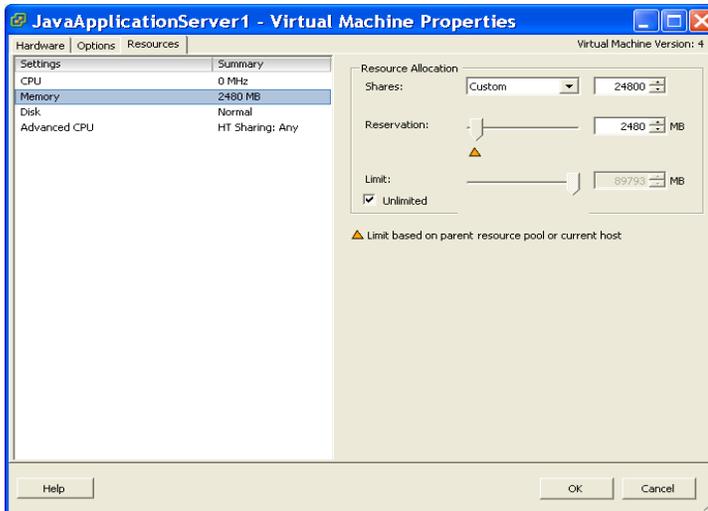
Other JVM options exist that can be used to determine the thread stack size. An example of this option for the Sun JVM is:

-XX:ThreadStackSize=<size in KB> where KB is 1024 bytes

By sizing the virtual machine memory correctly for the processes within it, you avoid the occurrence of guest operating system swapping due to memory pressure. If the guest operating system decides to swap out any part of the memory pages that make up the JVM heap, then the Java application's performance will be affected. If swapping is occurring, it can be seen using the operating system's own tools, such as "sar" or "vmstat" on Linux systems or "perfmon" on Windows systems. Re-size your virtual machine until no swapping is occurring in the guest operating system.

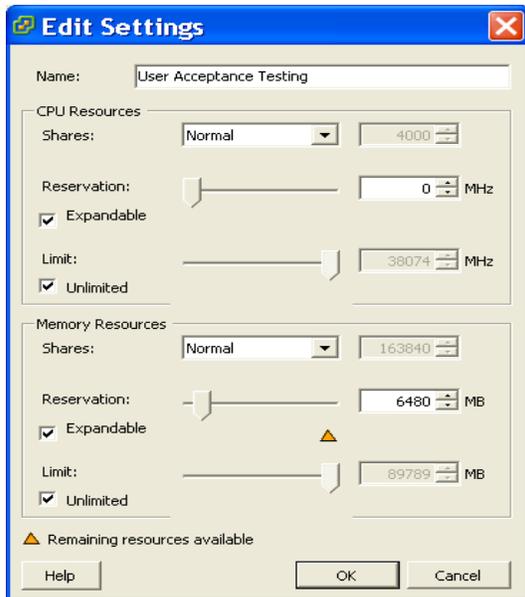
### 4.2.2. Memory Reservation

Set the memory "Reservation" value in the VI Client to the size of memory for the virtual machine. In Figure 3, the memory reservation is set to 2480 MB – ensuring that this virtual machine will always be allocated this amount of memory on any ESX server that it runs on.



**Figure 3: Adjusting the memory reservation for a virtual machine in the VI Client tool.**

You can also use the resource pool functionality in VMware DRS to safeguard the memory allocated to your virtual machines that host Java processes. By placing the virtual machine in a resource pool that has sufficient memory allocation to cater for the requirements, DRS will ensure that the virtual machine is placed on the correct ESX host server within the cluster. Set the **memory reservation for the resource pool** using the “Edit Settings” option on that item as shown in Figure 4. Here, a resource pool named “User Acceptance Testing” has a memory resource reservation of 6480 Mb, ensuring that it will have that much memory available for its resident virtual machines across a number of servers in the DRS cluster.



**Figure 4: Edit Settings for a Resource Pool**

### 4.2.3. Large Memory Pages

Large memory pages help performance by optimizing the use of the Translation Lookaside Buffer (TLB) where virtual to physical address translations are performed. Use large memory

pages as supported by your JVM and your guest operating system. The operating system and the JVM must be informed that you wish to use large memory pages, as is the case when using large pages in physical systems. Check the documentation for your guest operating system to determine how large pages are enabled for it (see reference 8 for more detail). To enable large memory pages in the JVM, use the following options:

- XX:+UseLargePages for Sun JVMs from version 5.0 onwards
- Xlp for the IBM JVM

### 4.3. ESX Memory

The hardware server controlled by ESX is equipped with a certain amount of physical memory, or RAM. That memory is used to hold the ESX code itself as well as the virtual machines that execute on it. Usually those virtual machines that host a Java process have a sizeable memory allocation. If ESX cannot find enough free memory to accommodate the needs of its resident virtual machines, it first invokes the balloon driver in the guest operating system of one or more virtual machines. If this fails to free up enough memory, ESX itself begins to swap at the host level, negatively impacting the overall performance of the system. This ESX level swapping should be avoided by either adding more physical memory to the ESX host server or by moving one or more virtual machines to other ESX hosts that have spare memory and CPU capacity. The latter can be done automatically by organizing your DRS resource pools correctly for your loads.

#### 4.3.1. Memory Overhead

ESX has three separate areas of memory overhead:

- A fixed system-wide overhead for the service console (272 Mb for ESX 3.x).
- A fixed system-wide overhead for the VMkernel part of ESX. This overhead depends on the number and size of the device drivers contained in it.
- An additional overhead for each virtual machine. The virtual machine monitor (one for each virtual machine) requires a certain amount of memory for its code and data.

The fixed system-wide overhead for ESX can be seen in the VI Client tool. For a particular ESX host, select the Configuration tab and choose the “Memory” menu item to see these values, as shown in Figure 3. You can see the various memory components in the panel on the right-hand side.

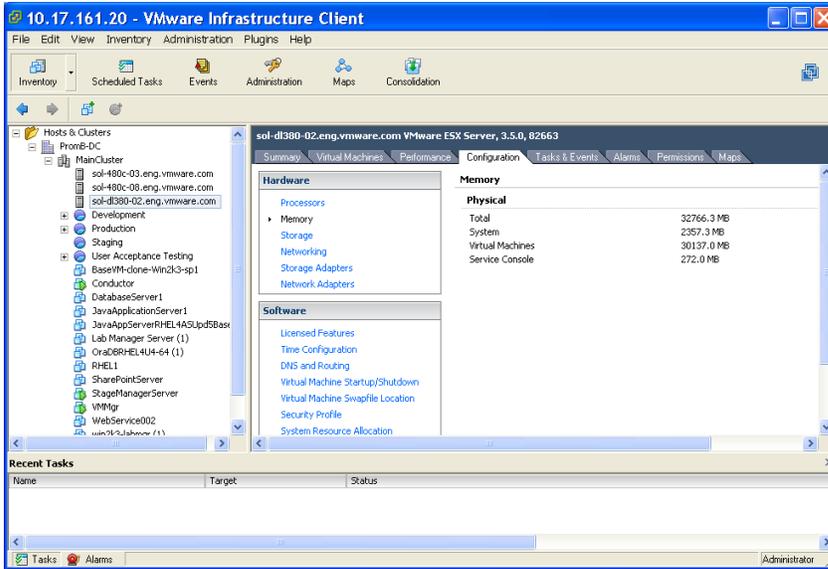


Figure 6: Memory values for an ESX host in the VI Client tool.

In the example in Figure 6, the host server’s total physical memory is 32Gb, of which 30Gb is available for virtual machines to use after system overheads are subtracted. The additional overhead for each virtual machine depends on the number of virtual CPUs assigned to it and on whether it is a 32-bit or 64-bit virtual machine. Chapter 9 of the VMware Resource Management Guide [Ref 9] provides details on these overhead values.

**ESX Memory Summary:**

- Appropriate sizing ensures that no swapping or memory contention occurs between each of the virtual machines running Java and other virtual machines on the same server.
- Use VMware Distributed Resource Scheduler (DRS) to balance memory use across machines.

**4.4. Examining Memory Consumption**

Two tools are very useful in looking at memory: the VI Client tool and “esxtop.” The VI Client tool can use an individual ESX instance or vCenter Server as its source of performance data. The esxtop tool is invoked from the ESX service console and it can be used interactively or in batch mode. For ESX 3.5 onwards, the command

```
esxtop -a -b > outputfile.csv
```

generates esxtop data in the outputfile.csv file that captures all columns of data (-a) in batch mode (-b). Use the -n <iterations> option to esxtop if you want to run it for a set number of samples and then stop. The data can then be examined offline. Table 1 shows the main memory-specific counters to examine to ensure the various memory areas are in a healthy state.

Table 1: Measurements to monitor in the “esxtop” tool and the VI Client tool for memory issues

Activity	esxtop columns	VirtualCenter / VI client
ESX Host Swapping	SWR/s (reads per second) SWW/s (writes per second)	mem.swapin, average mem.swapout, average
Guest OS Swapping	MCTLSZ	Mem.vmmemctl, average

### Note on Memory Retention Problems

These types of problems arise when memory is consumed in the Java process by objects that are not needed, but they are not garbage collected because there is an outstanding reference to them. These retained objects in memory can grow to a point where the heap memory is exhausted because of them, which can eventually cause an “OutOfMemoryException” at runtime. Memory retention conditions (sometimes called “Memory Leaks”) are really caused by an error in the application or application infrastructure implementation code. These programs behave in the same manner on virtual as on physical machines. These types of errors should be eliminated before the Java application system is deployed.

## 5. Virtual CPUs and Threads

Threads are part of the Java runtime environment that allow the parallel execution of segments of code. Threads may be created for the JVM’s own purposes or by the user’s Java code, or they may be created by a container technology such as a J2EE application server that uses them on the user program’s behalf. Threads are often created in thread pools where they may be re-used over time.

There may be many threads alive within a Java process at one time. It is not uncommon to have over twenty threads at once in even small Java programs (as the JVM requires some threads of its own, such as one or more garbage collector threads). Although there appear to be many threads alive at the same time, often only a small subset of them are in the “running” or “runnable” state at any one time. Many threads spend considerable time in the “idle,” “waiting” or “blocked” state, where they may be waiting for data to appear on a socket or some other condition before they are ready to run.

### 5.1. Matching Threads and Virtual CPUs

On physical operating systems, the collection of JVM threads can be spread out across all the physical CPUs available to the operating system, if they need to operate in parallel. In virtual machines with one virtual CPU configured, ESX will give each runnable thread a time-slice on that virtual CPU. When multiple virtual CPUs are configured in a virtual machine, ESX can distribute the runnable application threads to as many of the virtual CPUs as it needs to.

In the virtual machine implementation, if there are several threads that are ready to run at the same time, then that application system **may or may not** benefit from having multiple virtual CPUs present in the virtual machine. The nature of the work being done in the threads really determines the performance gain from parallel thread execution. If each thread takes up only a small portion of its allotted time slice, then a single virtual CPU may be as good as or better than multiple virtual CPUs. This really needs to be thoroughly tested with your particular application under suitable loads to establish the best virtual CPU configuration.

### Virtual CPUs and Threads for Garbage Collection

JVMs have options that allow the user to determine the number of garbage collection threads that may be active at any one time. This feature is determined by the following JVM runtime options:

- Xgcthreads<n> (for the IBM JVM)
- XXgcthreads<n> (for the JRockit JVM)
- XX:ParallelGCThreads=<n> (for the Sun JVM)

where  $\langle n \rangle$  represents the number of GC threads to be used by the JVM. If the number of virtual CPUs configured in the virtual machine containing a Java program is not equal to or greater than the number of Java GC threads, then the performance gains that are expected from using multiple GC threads will be affected. Since in that case there are not enough virtual CPUs to schedule all of the GC threads at once, then some of the GC threads will be held up and the time to complete GC events will likely be longer.

## 5.2. Virtual CPU Recommendations

One thread may execute on one virtual CPU at any one time. Determine the optimum number of **virtual** CPUs for a virtual machine that hosts a particular Java application by testing the application in the virtual machine with one, two, or more virtual CPUs at different times. These tests should each be executed with the same user load, while at the same time measuring application throughput and response time. The best match of virtual CPU setup to the application type is very difficult to predict without this application testing. Some multi-threaded applications can behave better on a virtual machine with one virtual CPU, whereas others gain performance benefits from having multiple virtual CPUs available.

## 6. Disk I/O

Java applications may perform considerable amounts of I/O, depending on the task that they are carrying out. The performance of these operations will be limited by the latency of the I/O device that is being written to or read from. In many cases, Java processes write to their log files on disk – for example, to store error and warning messages. They may also write to other data files on disk. Many J2EE applications make use of JDBC connection pools, via Enterprise Java Beans (EJBs) or otherwise, to read and write data to remote databases. The use of these JDBC connection pools cuts down on the amount of local I/O performed by the Java process but increases its network I/O. You should check that the I/O latencies of your ESX host server system are appropriate for the applications being hosted on them.

### 6.1. Disk I/O Recommendation

Check the guest average (GAVG) and disk average (DAVG) latency time columns in the **esxtop** tool's output. Ensure that these latencies are within the expected ranges.

## 7. Timekeeping

Timekeeping can be different in virtual machines than on physical machines for a variety of reasons, as explained in reference 7. Timekeeping can have an effect on Java programs if they are sensitive to accurate measurements over periods of time, or if they need a timestamp that is within an exact tolerance (such as a timestamp on a shared document or data item). For this reason, you will need to perform certain actions to synchronize the timing inside the guest operating system in a virtual machine with that of the outside world. VMware Tools contains features that are installable into the guest operating system to enable time synchronization and the use of those tools is recommended. The effects of timer interrupts are also discussed in this section as the frequency of those interrupts can have an effect on the performance of your Java application.

### 7.1. Timekeeping Recommendations

1. Synchronize the time on the ESX host with an NTP source. See Ref. 7 for details on this.
2. Synchronize the time in the virtual machine's guest operating system

- for Linux guest operating systems using an external NTP source (see Ref. 7)
  - for Windows guest operating systems using **w32time**
3. Lower the clock interrupt rate on the virtual CPUs in your virtual machines by using a guest operating system that allows lower timer interrupts (Examples of such operating systems are RHEL 4.7 and later, RHEL 5.2 and later and the SuSE Linux Enterprise Server 10 SP2). See reference 15 for more information on timekeeping best practices for Linux.
  4. Use the Java features for lower resolution timing that are supplied by your JVM, such as the option for the Sun JVM on Windows guest operating systems:
    - XX:+ForceTimeHighResolution
 You can also set the `_JAVA_OPTIONS` variable to this value on Windows operating systems using the technique given below (in cases where you cannot easily change the Java command line, for example).
  5. Use as few virtual CPUs as are practical for your application, based on the results for application-specific performance testing as described in the “Virtual CPUs and Threads” section above. The more virtual CPUs there are in your virtual machine, the harder ESX has to work to provide regular timer interrupts to them.
  6. Avoid using the `/usepmtimer` option in the `boot.ini` system configuration for Windows guest operating systems that use an SMP HAL.

### Setting the Sun JVM Option

To set the `_JAVA_OPTIONS` environment variable:

1. Click **Start > Settings > Control Panel > System > Advanced > Environment Variables**.
2. Click **New** under System Variables. The variable name is `_JAVA_OPTIONS`. The variable value is **-XX:+ForceTimeHighResolution**.
3. Reboot the guest operating system to ensure the variable is propagated properly.

## 7.2. Effects of Timer interrupts

Higher resolution timer interrupts (e.g. 1000 Hz) cause more work to be done by ESX on behalf of a virtual machine than lower resolution timers do (e.g. 100 Hz). The best practice for better performance is to use the lowest acceptable timer interrupt resolution. The guest operating system determines the timer interrupt rate in the virtual machine. Using the Sun JVM on Windows operating systems, for example, a Java program that has a `Thread.sleep()` call with an argument value that is not a multiple of 10ms will change the time resolution for that guest OS to be 1ms. This higher resolution of timing interrupts causes more performance overhead. The option for the Sun JVM named

`-XX:+ForceTimeHighResolution`

is designed to force this higher timer resolution behavior, but due to a bug in the JVM this option causes the opposite behavior, i.e. the timer resolution is never set to 1ms. For more details on this issue, see references 12 and 13.

Windows-based machines have a default timer interrupt period of 10ms, although some systems have a 15.625 ms period. When running in a virtual machine, Windows almost always uses the 15.625 ms period.

This timer interrupt period may be modified by programs using the Windows APIs. A program such as the JVM can request a 1ms timer interrupt period. The Sun JVM uses this 1ms period to allow for higher resolution `Thread.sleep()` calls than would otherwise be possible. The example program given below causes this higher interrupt rate to be used. It invokes the

*Thread.sleep(Integer.MAX\_VALUE)* method that causes the guest operating system to switch to a 1ms period for the duration of the sleep (because the value is not a multiple of 10ms).

```
public class Sleeper {
    public static void main(String[] args) throws Throwable {
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

You can see the interrupt period being used in a Windows operating system using the perfmon tool. Within perfmon, add a new item to monitor by clicking the + icon above the graph image. Select the "interrupts/sec" item from the list and add it. Then right click on "interrupts/sec" under the graph and edit its properties. On the "data" tab, change the "scale" to 1 and on the graph tab change the "vertical scale" maximum to be 1000. Let the system run for a few seconds and you will then see the graph drawing a steady line. If you have a 10ms interrupt occurring then it will be approximately 100, for 1ms it will be close to 1000, for 15.625 ms it will be roughly 64, etc. All the above numbers are close approximations, as there are other sources of interrupts present also.

Any application can change the timer interrupt period to affect the whole guest operating system. Windows only allows the period to be shortened, thus ensuring that the shortest requested period across all applications is the one that is used. If a process does not reset the period then Windows resets it when the process terminates. The JVM does not arbitrarily change the interrupt rate at start time – although it could do this. The reason it does not do so is that there is a potential performance impact to everything on the system due to the increase in clock interrupts. Other applications may also change the interrupt rate. A browser running the JVM as a plug-in can also cause this change in interrupt rate if there is an applet running that uses the *Thread.sleep()* method in a similar way to the example program. Furthermore, after Windows suspends or hibernates, the timer interrupt is restored to the default value, even if an application using a higher interrupt rate was running at the time of suspension/hibernation.

If your Java program is executing on a Windows platform and it is doing timed waits or sleeps at a resolution other than multiples of 10ms, then when using the Sun JVM you should execute the JVM with the **-XX:+ForceTimeHighResolution** option. This option has a flaw in its implementation that disables the internal attempts to use the high-resolution timer for *Thread.sleep()* calls (i.e. it reverses the behavior that its name implies). If you omit this flag, and if you set an interrupt period other than 1ms, where the requested delay is not a multiple of 10ms, then the internal sleep implementation will change it to 1ms. This higher resolution clock timer interrupt period can decrease the performance of your application in a virtual machine.

### 7.3. The /usepmtimer option with Windows Guest Operating Systems

Problems related to timers have been found in Windows guest operating systems when the "boot.ini" configuration contains the /usepmtimer option. Avoid specifying /usepmtimer in the boot.ini configuration when using Windows 2003 or Windows XP with an SMP HAL. A VMware Knowledge Base article containing details on this subject is available here: <http://kb.vmware.com/kb/1011714>.

## 8. Micro-benchmarks

The phenomenon described here applies equally to running Java in virtual machines as it does for running physical Java programs. However, since performance is often being examined very closely on virtual machine platforms, it is worth repeating here. Running performance-related tests on very small Java programs that may seem to represent your application's code may not deliver accurate results – either on physical implementations of Java or on virtual machines running Java. These smaller programs are sometimes chosen as extracts from the main application itself, but it is better to use the full application code for testing purposes.

### 8.1. Warm-up Time

Give your Java program an adequate period of time to “warm up” before you take any performance-related measurements. The JVM takes some time to load classes. It also takes time to iterate through the bytecodes and decide whether to compile certain sections using the internal runtime compiler. For these reasons, we strongly recommend that you use your actual application code and an initial warm-up period in sizing exercises, rather than using representative subsets of your code, as the results from testing with these smaller subset programs can be misleading.

## 9. Conclusions

In the majority of cases, no change to the setup of a Java process is required to achieve good performance when running it on a virtual machine based on VMware ESX. There are many examples of this in the user base today. To ensure best application performance, it is wise to look at the memory, CPU, disk I/O and timekeeping configuration of the virtual machine to ensure that the Java process is optimized for performance.

This paper suggests some health checks and configuration settings that can be done at the ESX and virtual machine levels to ensure the correct setup. These checks should be performed in concert with your regular Java tuning optimizations that apply on physical implementations of Java processes – all the same rules that you use in the physical world still stand. VMware customers who follow these guidelines are already experiencing the considerable benefits, performance and cost savings that result from virtualizing their Java implementations today.

## 10. References

1. "Java Performance Tuning", J. Shirazi, published by O'Reilly Press
2. "Performance Analysis for Java Web Sites", S. Joines, R. Willenborg, K. Hygh, published by Addison-Wesley
3. "Enterprise Java Performance", S. Halter, S. Munro, published by Prentice Hall
4. The VMware Performance Community  
<http://communities.vmware.com/community/vmtn/general/performance>
5. Java Support for Large Memory Pages  
<http://java.sun.com/javase/technologies/hotspot/largememory.jsp>
6. Garbage Collection Policies for Java  
<http://www.ibm.com/developerworks/java/library/j-ibmjava2/>
7. Timekeeping in VMware Virtual Machines  
[http://www.vmware.com/pdf/vmware\\_timekeeping.pdf](http://www.vmware.com/pdf/vmware_timekeeping.pdf)
8. Large Page Performance: VMware Performance Study  
[http://www.vmware.com/files/pdf/large\\_pg\\_performance.pdf](http://www.vmware.com/files/pdf/large_pg_performance.pdf)
9. VMware Resource Management Guide  
[http://www.vmware.com/pdf/vi3\\_35/esx\\_3/r35u2/vi3\\_35\\_25\\_u2\\_resource\\_mgmt.pdf](http://www.vmware.com/pdf/vi3_35/esx_3/r35u2/vi3_35_25_u2_resource_mgmt.pdf)
10. "Java Threads", S. Oaks and H. Wong, published by O'Reilly Press.
11. Sun JVM runtime options  
<http://blogs.sun.com/watt/resource/jvm-options-list.html>
12. Sun Developer Network entry on -XX:ForceTimeHighResolution  
[http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=6435126](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6435126)
13. Inside the Hotspot VM: Clocks, Timers and Scheduling Events  
[http://blogs.sun.com/dholmes/entry/inside\\_the\\_hotspot\\_vm\\_clocks](http://blogs.sun.com/dholmes/entry/inside_the_hotspot_vm_clocks)
14. VMware Performance Blog  
<http://blogs.vmware.com/performance/>
15. Timekeeping Best Practices for Linux  
<http://kb.vmware.com> (search for article number 1006427)

## 11. Appendix 1 : Checklist for Issue Resolution

- What vendor's Java technology is being used? What is the version of the JVM?
- Are there any special parameters being supplied to the JVM at runtime, such as -Xms and -Xmx to specify heap sizes?
- Where Java is only one part of the system, have you profiled the entire system to uncover system-wide causes of problems?
- Is there a measure of both throughput and response time for the system?
- Is more than one database being used at the back end of the system to answer any one query or update?
- Is more than one Java process executing in one virtual machine?
- Is the virtual machine's guest operating system swapping?
- Is the ESX host server swapping?
- How much memory is allocated to the Java heap?
- What is the frequency and duration of occurrences of each type of garbage collection?
- How much extra memory is available in the guest operating system beyond that given to the Java heap?
- Is the application multi-threaded?
- Is there more than one thread that is ready to run at any one time?
- Does the number of concurrently running threads exceed the number of virtual CPUs configured in the virtual machine (e.g. garbage collector threads)?
- What are the disk and network latencies for the virtual machine as seen in the "esxstop" tool?
- Can the timer resolution in the guest operating system be lowered in value without affecting the Java program?
- Check that the /usepmtimer option is NOT present in the boot.ini file for Windows guest operating systems with SMP HAL. If it is present, remove it and reboot the guest.



**VMware, Inc.**  
**3401 Hillview Ave Palo Alto, CA 94304 USA**  
**Tel 650-427-5000 Fax 650-427-5001 [www.vmware.com](http://www.vmware.com)**

© 2009 VMware, Inc. All rights reserved. Protected by one or more of U.S. Patent Nos. 6,397,242, 6,496,847, 6,704,925, 6,711,672, 6,725,289, 6,735,601, 6,785,886, 6,789,156, 6,795,966, 6,880,022, 6,961,941, 6,961,806, 6,944,699, 7,069,413; 7,082,598 and 7,089,377; patents pending.

