Tuning garbage collection for Java and WebSphere running classic JVM on i5/OS

Jeremy Arnold
Scott Moore
IBM Business Strategy and Enablement
February 2006

Table of contents

.

Abstract

This paper provides performance tuning tips, techniques, and tools for Java applications that run on the IBM System i (System i5, eServer i5, and iSeries) platform using the classic Java Virtual Machine (JVM). In particular, this paper focuses on tuning methods that affect the JVM heap management environment. Code and command examples, as well as screen captures related to the tools discussed in this paper, are included for the reader's better understanding.

Introduction

Well-performing Java™ applications tune the amount of overhead that the Java heap management has on the Java Virtual Machine (JVM™). You can do many things to minimize this overhead, from changing the application, adding additional memory, or modifying Java options. This paper will focus primarily on modifying the Java options for the best performance. It will focus on the following areas:

- Explaining what the garbage collector does inside a JVM and how it does it
- Examining the tools available to analyze your garbage collector performance
- Discussing the various ways to tune for the optimal use of CPU and memory

Very Important: Starting in IBM® i5/OS® Version 5 Release 4, two JVMs are available:

1. The classic JVM, which IBM introduced in 1998 on the IBM AS/400® platform, is a 64-bit JVM. The classic JVM includes support for the Java Software Development Kit (SDK™) 1.3, 1.4, and 1.5 (Java 5.0).
2. An entirely new JVM, called IBM Technology for Java (32-bit) is also available to run on i5/OS V5R4. The new JVM currently supports JDK 1.5 (Java 5.0) and offers significant performance improvements for certain types of applications.

Tuning recommendations are dependent on which JVM you are using. This paper is intended for use only if you are using the classic JVM. You can determine if you are running the classic JVM by doing the following:

1. If you are running on i5/OS V5R3 or earlier, you are running the classic JVM.
2. If you are running on i5/OS V5R4, display the job log of the JVM by performing the i5/OS Work with Active Jobs (**WRKACTJOB**) command and selecting **5 – Work with** next to the JVM. On the Work with Job screen, select **10 – Display job log**. You will see a screen that looks like Figure 1:

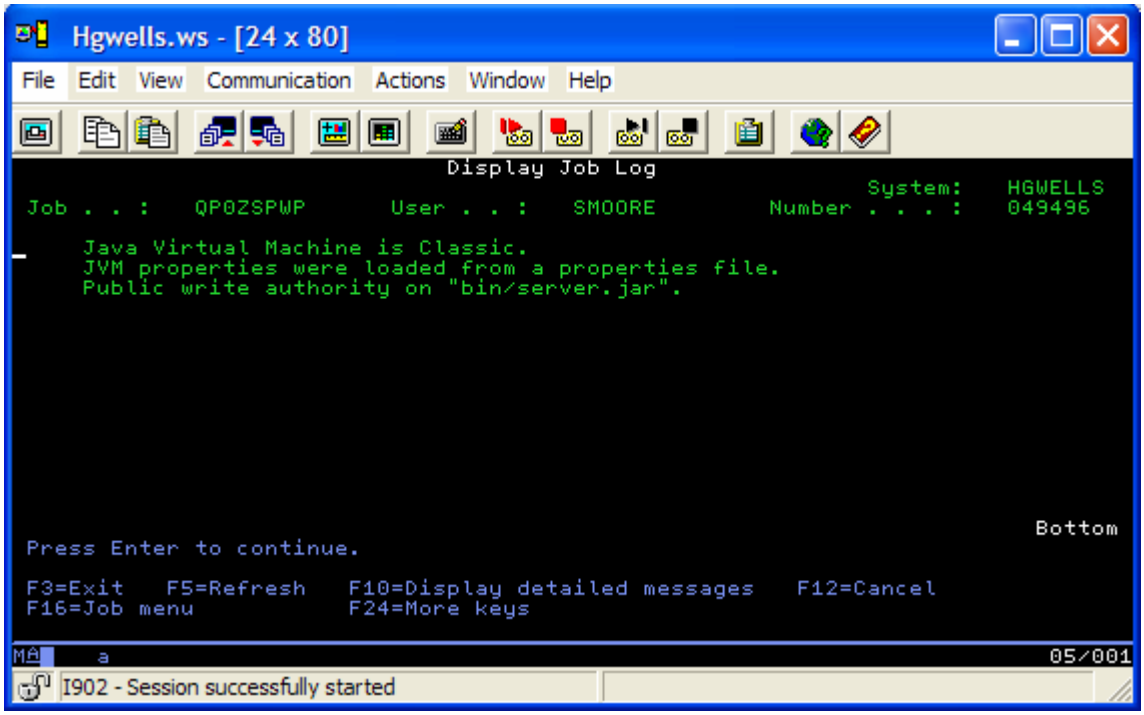

Figure 1: The Java virtual machine is running classic JVM

The first line is an indication that the JVM is indeed running the classic JVM.

See the *V5R4 Performance Capabilities Reference Manual for V5R4* to help decide if you can use the classic JVM or the IBM Technology for Java JVM. If running IBM WebSphere® V6.0 and earlier, you are definitely running the classic JVM.

Understanding Java garbage collection

When running in a JVM, many Java objects are created. During the normal flow of a program, some of these objects will become unused. It is up to the garbage collector (GC) to free these unused objects up, so that the JVM can reuse their memory.

The GC runs in cycles. For example, the garbage collector can run for two seconds, go idle for 30 seconds, and then will run again for two seconds. The idle time depends on how many objects are being created within the JVM. The iSeries GC is asynchronous (runs in the background), concurrent (runs at the same time as the rest of your application), and multithreaded. While these features normally improve GC performance, they also make it more difficult to determine if the garbage collector is not tuned properly. Though JVMs on other platforms might experience long pause times with improper GC tuning, the iSeries collector does not typically cause long pauses in your application threads. Therefore, you might not realize that your application is experiencing problems with GC unless you specifically look for these problems.

Because nearly all Java or WebSphere applications can benefit from tuning GC, examine GC performance when deploying new applications or when the system load or hardware

configuration for existing applications change. It is also wise to examine the GC if your applications are experiencing performance problems.

Two parameters influence the GC when running the classic JVM on the System i platform: the initial heap size and the maximum heap size. You can set the initial heap size with the **-Xms** (or **-ms**) variable on the Java command line. For example, the **java -Xms256m MyClass** command will create a 256-megabit initial heap size. Similarly, you can set the maximum heap size with the **-Xmx** or **–mx** variable. You should almost never change the maximum size from the ***NOMAX** default value. In WebSphere Application Server V5.0 and beyond, you can set the initial heap and max heap size using the administration console. Shown in Figure 2 is a screen capture of the WebSphere V6.0 Administration Console. Notice where you can set the value:

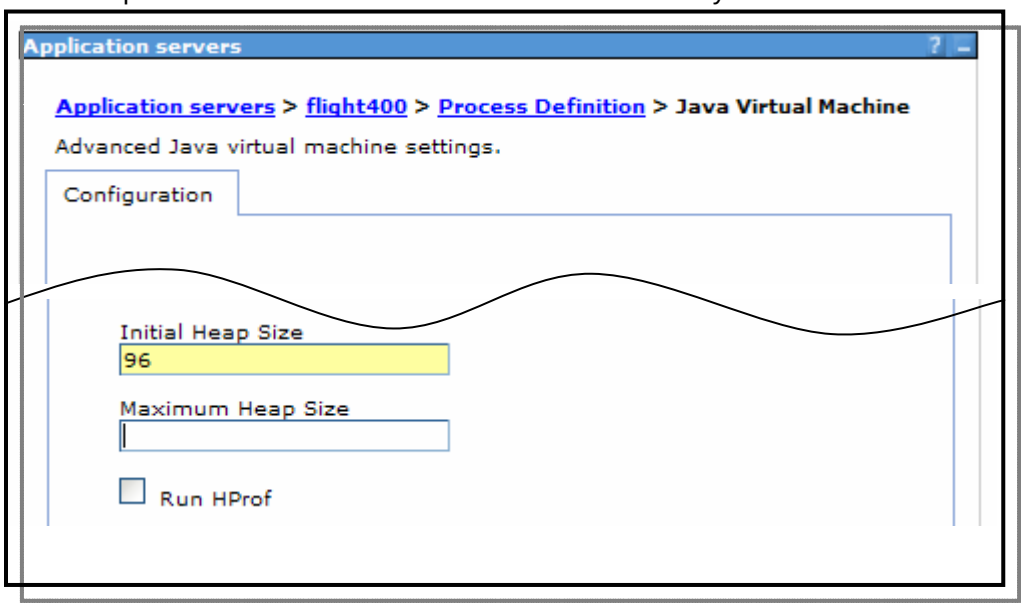

Figure 2: Advanced JVM settings

(Note: The ***NOMAX** value for the maximum heap size in the WebSphere platform is represented by **0** or a blank field.)

While other platforms also have these parameters, the behavior of running the classic JVM on the System i5 platform is unique. For instance, the initial heap size is more accurately called the GC threshold." After the new memory allocation of the JVM reaches this threshold, the garbage collector will run. After reallocating the initial heap size amount, the GC runs again. Therefore, increasing this value makes the GC run less frequently and allows the heap to grow larger between collections. This, in turn, makes each GC cycle take longer, because it must scan more memory. Setting the threshold to its optimal value involves finding the right balance between the frequency that the collector runs and the heap size. If it is set too low, the GC runs too frequently, resulting in wasted CPU cycles. Setting it too high allows the heap grow too large, which can result in a higher cache miss ratio and increased paging. In some rare cases, the application can even allocate memory faster than the GC can collect it. This can result in heap growth over time until the JVM runs out of memory.

Using tools to analyze and tune the GC

A few tools are available that can give an indication of how well the GC is running in the JVM. In this section, you will read about three of these tools.

PEX TPROF trace

One of the easiest ways to determine if the GC is running efficiently is to measure the percentage of CPU time spent performing garbage collection. You can do this by using a Performance Explorer (PEX) trace profile (TPROF). (For details, see **Collecting and analyzing an IBM iSeries TPROF trace**. A link to this site is available in the **Resources** section of this paper.) You can run a TPROF trace against any JVM, without any kind of restart required or special JVM properties passed into the JVM. This paper will go through the steps of creating a TPROF trace.

When you have collected the trace profile data, you can use one of the following tools to view this data:

- IBM Performance Trace Data Visualizer (PTDV) is a free tool. (You can download this tool from the IBM alphaWorks® Web site.)
- Alternatively, you can print the report using the Print PEX Report (**PRTPEXRPT**) command with the ***PROGRAM** option. (This command is available with the Performance Tools licensed program product [LPP], 5722PT1.) The bottom section of the report shows the percentage of CPU time spent in the various programs on the system. The Java garbage collection (**JAVAGC**) program indicates the time spent in garbage collection. In Figure 3 below, about 5.4% of the CPU time is performing garbage collection.

```
Histogram Hit Hit   Cum            Start Map    Stmt Name
          Cnt %       %             Addr Flag Nbr
*****   11908 14.6 14.6 0F147E886E1F39C4 ==   0 WEBASADV4_
**       6113  7.5 22.2 0AE07FA0D520DB08 ==   0 JAVA400_JD
**       5041  6.2 28.4 2FDF43EB09180D40 ==   0 LIB_QP_EJB
*        4359  5.4 33.7 FFFFFFFFB39D7CC4 ==   0 JAVAGC
*        4292  5.3 39.0 FFFFFFFFFE9C6280 ==   0 JAVADEEP
         2215  2.7 41.7 2E609B9CDE1EB93C ==   0 JAVA_EXTLI
         1946  2.4 44.1 03E126CBCE10EF30 ==   0 EXT_QP_DB2
```

Figure 3: Sample trace profile output

As a rule of thumb, the time spent in Java garbage collection will be less than 10% of the total CPU time. It is best to have it around 3%, but this might not always be possible. If the percentage is too high, there is probably room for improvement.

If the amount of time spent in garbage collection is too high, additional analysis is often unnecessary. You can just try increasing the initial heap size, as explained in the **Tuning the GC** section. Then monitor your application's performance (throughput and response time) to ensure that the performance improves.

You can also collect another trace profile to see if there is additional room for improvement.

Verbose garbage collection

At times, it might be helpful to collect additional data about the GC as its running. You can accomplish this by enabling verbose garbage collection. This will dump several pieces of information into **System.out** file each time the GC runs. This includes the current heap size, as well as the number and size of objects collected, number of objects in the heap, amount of time the collector ran, and other information.

You can enable verbose garbage collection by including the **-verbosegc** parameter on the Java command line. Alternatively, in WebSphere V6.0, you can use the **Enable garbage collection verbose mode** check box from the WebSphere Administration Console (Figure 4). Both of these require that you restart the JVM after making this update.

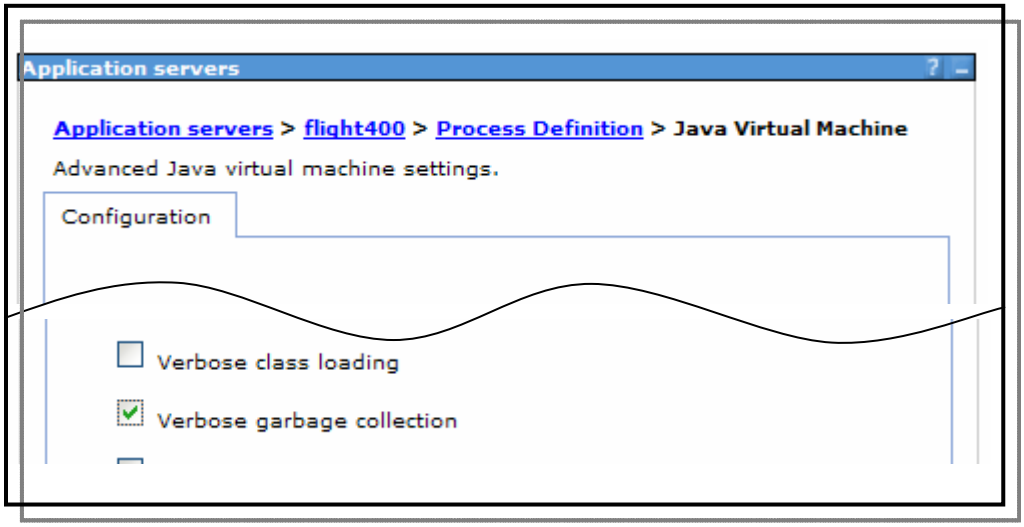

Figure 4: Shows how to set verbose garbage collection in the WebSphere V6.0 Administration Console

Verbose garbage collection will dump some additional text to the **stdout** file. Below, in Figure 5, is an example of some output that verbose garbage collection dumps into the file.

```
GC 4: starting collection, threshold allocation reached.
GC 4: live objects 2562187; collected objects 4936351; collected(KB) 541840.
GC 4: queued for finalization 0; total soft references 92; cleared soft references 5.
GC 4: current heap(KB) 1171424; current threshold(KB) 524288.
GC 4: collect (milliseconds) 4138.
GC 4: current cycle allocation(KB) 236160; previous cycle allocation(KB) 524314.
GC 4: total weak references 684; cleared weak references 0.
GC 4: total final references 11797; cleared final references 63.
GC 4: total phantom references 0; cleared phantom references 0.
GC 4: total old soft references 0; cleared old soft references 0.
```

Figure 5: Sample verbose garbage collection output

For the purposes of tuning the GC, the most important of these fields are:

- **GC 4:** The fourth GC cycle since the JVM started
- **Live objects:** Number of objects currently active in the JVM
- **Collected objects:** Number of objects collected during this cycle
- **Collected(KB):** Total size of the objects collected during this cycle

- **Current heap(KB):** The current heap size
- **Current threshold(KB):** The threshold value (Also referred to the initial heap size)
- **Collect (milliseconds):** Elapsed time for this cycle
- **Current cycle allocation(KB):** Memory allocated since the current cycle began
- **Previous cycle allocation(KB):** Memory allocated since the last cycle began

The current threshold is the value set for the initial heap size (512 megabytes in the example output). The previous cycle allocation is normally close to this value, because the GC cycle is triggered when the amount of memory allocated since the last cycle begins to reach the threshold value. The example output shows that the GC cycle took more than four seconds to complete. During that time, the current cycle allocation reached more than 200 megabytes. This is about 40% of the threshold value, which suggests that the total time between the beginning of this cycle and the next cycle is around 10 seconds. This cycle collected nearly 5 million objects (collected objects), leaving only 2.5 million objects in the heap at the end of the cycle (live objects).

In general, it is best to have a low cycle time. One to two seconds is ideal, but times of five to 10 seconds are common for WebSphere applications, especially on older or slower hardware. It is also best to have some time between collection cycles (in other words, current cycle allocation will mostly likely be less than the current threshold). These two goals work against each other. Increasing the threshold value allows the heap to grow, which results in more time between cycles but lengthens each cycle. Decreasing the threshold shortens each cycle but also reducers the time between cycles.

The key to tuning the GC is to find a balance between these two goals. This is why examining the CPU consumed by the GC is generally better than looking at values such as the current heap size.

DMPJVM

Another tool that you can use to learn about garbage collector performance is the Dump JVM (**DMPJVM**) command. This tool provides a spool file with information about your JVM, including some of the key GC data (initial heap size, maximum heap size, current heap size, and number of collections since the JVM started). The **DMPJVM** spool file also includes a dump of the objects currently in the heap, which can be helpful for analyzing object leak problems. (Object leaks occur when your application creates new objects and keeps a reference to the objects even when they are no longer needed. This prevents the GC from collecting them. In the **Resources** section, refer to **Collecting and Analyzing a Java object creation trace on the IBM eServer iSeries platform** for more details.) Figure 6 shows a sample of the GC section of **DMPJVM** output.

```
.....................................................................
. Garbage Collection .
.....................................................................
Garbage collector parameters
Initial size: 262144 K
Max size: 240000000 K
Current values
Heap size: 449952 K
Garbage collections: 278
Additional values
JIT heap size: 85728 K
JVM heap size: 186588 K
Last GC cycle time: 1302 ms
```

Figure 6: Sample DMPJVM output

While the DMPJVM data is only a snapshot and does not provide the details available with verbose garbage collection, you can run it without restarting the JVM. For this reason, it is useful for getting some information about the JVM after a problem has occurred. Details on the **DMPJVM** command can be found in the iSeries Information Center. A listing for this site is available in the **Resources** section of this white paper.

Tuning the GC

When running the classic JVM in the i5/OS environment, it is simple to tune the GC. There are relatively few parameters and values that you can tune to affect the garbage collector. In reality, you have two options to reduce the time spent in collecting objects:

1. Modify the GC tuning parameters.
2. Reduce the number of objects that your application creates.

Modifying the GC tuning parameters

Tuning your application's initial heap size is an iterative process. Begin by picking a reasonable starting point. The correct size varies depending on the system and application, but Table 1 provides a rule of thumb in choosing the starting size, based on the number of processors that the system utilizes. These guidelines assume that the Java or WebSphere application is the only significant application running on the system and that it is utilizing most of the CPU resources. You might need to adjust these suggestions for other conditions.

Processors	Initial heap size
1	96 megabytes
2	128 megabytes
4	256 megabytes
8	512 megabytes
12 or more	one gigabyte

Table 1: Suggestions for initial heap size

When you have picked a reasonable starting point, start your application and let it run for a time under the maximum load that you intend to handle, giving it time to reach a steady state (a few minutes is usually sufficient). It is best to use a load-generation tool to put a constant load on

your system. This allows you to tune your application in a development environment rather than a production environment.

In addition to providing a constant load (allowing you to see the effects of changes more accurately), this allows you to make changes as necessary without affecting users. While your application is running, use the aforementioned tools to measure the impact of your changes. Make sure that your load-generation tool also details the throughput and response time so that you can see the impact on your application's performance. Whenever you change performance parameters such as the GC threshold, you can measure the effects on throughput and response time to ensure that the changes actually help.

If the tools indicate that there is still room for improvement, change the threshold, and try a new run. In general, increase the threshold from the starting points in Table 1. When you have increased it too far, the throughput begins to degrade again, indicating that you must reduce the threshold.

These suggestions assume that your system has enough main store to handle larger heap sizes. In reality, this might not always be the case. In systems with limited memory, it might be necessary to set the GC threshold to a lower value, which increases collection frequency and decreases heap size. This allows the JVM to keep the entire heap in memory. Use the i5/OS Work with System Status (**WRKSYSSTS**) command or the IBM Collection Services tool (formerly IBM Performance Monitor) to monitor the non-database paging and faulting rates. If these rates get too high, the heap might be too large. (**Note:** The Collection Services provide the underlying performance, data collection function for the i5/OS environment. These services support the monitor and graph history functions and also provide input to the Performance Tools for iSeries.)

The definition of too large depends on a variety of factors: system size, number of disks, and system workload. However, sustained nondatabase paging rates greater than 10 faults per second by Java programs are generally cause for concern. Higher paging rates are acceptable during warm-up periods.

High paging rates might result from having the GC threshold set too high or might be a symptom of a larger problem. In this case, the first step is to isolate the JVM in its own memory pool. This reduces the effects that other applications might have on the JVM and makes it easier to identify whether the problem is with the GC settings, system configuration, or simply not enough hardware to handle the workload.

In cases where memory is especially limited, it might be useful to set the maximum heap size. Normally, this is left at the ***NOMAX** default value, which means that GC runs only when the GC threshold has been reached. If you do set a maximum heap size, the collector runs whenever the heap reaches that maximum size. However, unlike a normal garbage collection, if the GC reaches its maximum size, all application threads must wait until the collector has finished before they can continue running (a synchronous garbage collection). This results in undesirable pause times. Therefore, it is preferable to use the maximum heap size as a safety net to handle times of unexpected heap growth and ensure that the heap does not grow larger than the available memory. Be sure to set the GC threshold so that this maximum size is never actually reached under normal circumstances.

Reducing object creation

Tuning the collector is one way to reduce the amount of CPU cycle time spent in garbage collection. It is the easiest method, and it is best to try this before attempting other techniques. However, another way of reducing garbage collection time is to reduce the number of objects the Java code creates. If the number of objects your applications create is excessive, none of the tuning for the initial heap size will bring the CPU resources spent to a reasonable value.

With excessive object creation, there are two primary reasons that CPU utilization will be higher. First, the creation of an object will traverse code within the JVM heap management environment, as well as in the constructor for each object. Secondly, the GC will have to run more frequently to manage the objects as they are no longer needed, and that requires CPU resources. In some extreme cases, CPU usage by the GC can be more than half of the total JVM CPU usage.

The memory usage will be higher because there is more heap demand on allocating many objects in a short period of time, even if the objects are short-lived. Furthermore, the larger heap can have ramifications on other parts of the system, such as less efficient chip caching (L1, L2, and L3 cache).

Regardless of how many objects the JVM must handle, the GC can only deal with a maximum number of them at a time. If the number of object creations increases beyond the maximum, the heap can begin to swell in size because the GC will not be able to keep up with the number of creations. This is a simplified explanation, because there are many factors that determine the rate at which the GC manages objects. These other factors include:

- Object size
- Heap fragmentation
- Main store memory allocation of the JVM
- Other jobs using the memory

(See: **Collecting and Analyzing a Java object creation trace on the IBM eServer iSeries platform** in the **Resources** section for more information.)

Summary

The i5/OS implementation of the JVM provides advanced GC benefits for Java and WebSphere applications. However, the JVM requires some simple tuning efforts to optimize GC performance. Fortunately, you can usually accomplish this tuning by setting just one parameter, the initial heap size. The tools provided with the i5/OS operating system, the JVM, and WebSphere development environment offer multiple ways to analyze and monitor GC performance. In many cases, just a few minutes of analysis can result in significant performance improvements with no application changes.

Resources

These Web sites provide useful reference materials to supplement the information contained within this document.

- Performance Management for IBM eServer iSeries Web site:
  www.iseries.**ibm.com**/perfmgmt
- IBM AlphaWorks Performance Trace Data Visualizer:
  alphaworks.**ibm.com**/tech/ptdv
- Collecting and analyzing a Java object creation trace on the IBM eServer iSeries platform:
  ibm.com/servers/enable/site/education/abstracts/8d46_abs.html
- Java on iSeries performance guide for developers:
  ibm.com/servers/enable/site/education/abstracts/8f56_abs.html
- Collecting and analyzing an IBM iSeries TPROF trace:
  ibm.com/servers/enable/site/education/wp/9a1a/index.html
- Details on the **DMPJVM** command can be found in the iSeries Information Center at:
  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/cl/dmpjvm.htm

About the authors

Jeremy Arnold is a performance analyst specializing in Java and WebSphere performance. Jeremy has worked in the Rochester lab for six years, primarily working with Java performance. He has written numerous articles for trade magazines dealing with J2EE design that relates to Java performance.

Scott Moore is a senior performance analyst in the IBM Rochester lab and an i5/OS (OS/400®) and Java expert. Recently, Scott has been the team leader for WebSphere and Java performance on the System i platform and is currently helping solution providers with Java and WebSphere performance issues through the IBM eServer Solutions Enablement team.

Trademarks and special notices

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	eServer	i5/OS	DB2
ibm.com	iSeries	WebSphere	DB2 Universal Database
the IBM logo	AS/400	OS/400	System i
System i5			

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.