



**The Association of System
Performance Professionals**

The **Computer Measurement Group**, commonly called **CMG**, is a not for profit, worldwide organization of data processing professionals committed to the measurement and management of computer systems. CMG members are primarily concerned with performance evaluation of existing systems to maximize performance (eg. response time, throughput, etc.) and with capacity management where planned enhancements to existing systems or the design of new systems are evaluated to find the necessary resources required to provide adequate performance at a reasonable cost.

This paper was originally published in the Proceedings of the Computer Measurement Group's 2008 International Conference.

For more information on CMG please visit <http://www.cmq.org>

Copyright 2008 by The Computer Measurement Group, Inc. All Rights Reserved

Published by The Computer Measurement Group, Inc., a non-profit Illinois membership corporation. Permission to reprint in whole or in any part may be granted for educational and scientific purposes upon written application to the Editor, CMG Headquarters, 151 Fries Mill Road, Suite 104, Turnersville, NJ 08012. Permission is hereby granted to CMG members to reproduce this publication in whole or in part solely for internal distribution with the member's organization provided the copyright notice above is set forth in full text on the title page of each item reproduced. The ideas and concepts set forth in this publication are solely those of the respective authors, and not of CMG, and CMG does not endorse, guarantee or otherwise certify any such ideas or concepts in any application or usage. Printed in the United States of America.

10 WAYS TO SLOW DOWN YOUR JAVA APPLICATIONS

Peter Johnson
Unisys Corporation

You have recently released a new Java application into production and almost as soon as you do the complaints come in - it is running too fast. The application responds so quickly that your users and customers are feeling stressed out trying to keep up with it, and are asking if there is any way to slow it down. Well, you're in luck! This paper discusses 10 sure-fire ways to make that fast and responsive Java application slow down to a crawl.

Introduction

Don't you wish you had the problem mentioned in the abstract! Most likely, your experience has been the exact opposite – your Java applications tend to be slow and often experience unexpected response delays. What you really need are some practical ideas on how to go about improving the performance of those applications.

Well, you are in luck! This paper presents a top-ten list of practices that you can employ to speed up those sluggish applications. In typical top-ten fashion, the paper starts with practice #10 and works its way up to #1, with the number 1 practice providing the best overall performance gains.

#10: Don't Outguess the Compiler

This is the first of three practices that relate back to the early days of Java, when 1.0 was the current version. That version had a number of built-in limitations, and a number of performance issues that people had to work around.

Back before the HotSpot just-in-time (JIT) compiler was released, Java applications were run by an interpreter. This led to the following bit of wisdom:

Ancient myth #1: Examine the byte-code generated by the compiler and prefer Java coding practices that generate compact byte-code.

With the advent of HotSpot, and other modern JVMs that compile Java byte-code into machine language, this bit of wisdom is no longer valid. The actual code executed is not anywhere close to the byte-code produced by the compiler. In addition, every release of

the JVM adds in new JIT compiler techniques to make the resulting machine code even more efficient. Therefore attempting to guess what Java coding practice might result in more efficient code by examining byte-code is futile.

Instead, you should follow established coding practices that make the application more robust, extensible and maintainable. As an example, a typical Java coding practice is to make all fields private and then write public getters and setters for those fields. This is a good practice because it hides the actual implementation of the object behind the interface that the object presents. Someone concerned with performance might notice the extra byte-code generated by getters and setters and conclude that removing them, making the fields public, and accessing the fields directly will result in a performance boost. However, the HotSpot compiler is very clever and actually compiles standard getters and setters into the same machine code as direct field access.

A corollary to the above wisdom is to not compile with the debug flag (-g). The assumption is that this flag adds extra byte-codes to the compiled code, thus slowing it down. In fact, the debug flag simply adds extra metadata, such as line numbers, to the Java class file. None of that data affects the performance. However, having that data available can simplify the debugging of an issue in a running system. Therefore you should always compile with the debug flag set.

The lesson here is to not use the compiled byte-code as the coding standards arbiter. Rather, test the various constructs to see which perform better. And above all, realize that maintainability and extensibility play as big a role as performance in any application.

#9: Don't Outguess the Object Allocator or Garbage Collector

Another one of the tidbits of wisdom from the early days of Java went as follows:

Ancient myth #2: Object allocation and garbage collection are slow. Therefore, create a cache of objects and reuse them as needed.

For example, the application could create a cache of empty Customer objects. Then when it needs a Customer object, rather than having to create one, it gets one from the cache, initializes it, uses it, and then returns it to the cache when it is done. According to ancient wisdom, doing this avoids the cost of object allocation and reduces the work the garbage collector had to do.

However, the premises on which this myth was based are no longer valid. Object allocation is no longer a bottleneck. The locking issues associated with object allocation were solved a long time ago. In addition, you can use the `-XX:+TLAB` JVM argument to allocate to each thread its own area where objects can be created, eliminating the need for locking altogether.

In addition, hanging onto unused objects is guaranteed to put more strain on the garbage collector. Studies have shown that the vast majority of Java objects are short lived and never survive their first garbage collection [GC142]. Therefore the garbage collectors are coded to efficiently remove garbage. In fact, the garbage collector could be better termed a "live object collector" because that is what it spends its time doing: collecting the live objects. It ignores the dead objects. Therefore the more live objects it has to deal with, the longer it takes for it to do its job.

Having said all of this, there is one situation under which a cache is worthwhile – when initializing the object is expensive. Examples would include a database connection object (opening a connection to a database is a very time-consuming task) or a persistent object (where the data for the object comes from one or more tables in a database). In such cases a cache makes sense. However, application containers or frameworks often provide caches for such objects and you should use those caching mechanism rather than writing your own.

#8: To Synch or not to Synch

This final ancient practice is not a bit of wisdom, but rather a habit that programmers learned from Java 1.0 and still follow. You can easily spot such programmers

because their code often uses the classes Vector, StringBuffer and Hashtable.

The problem with these classes is that they are thread-safe. That means that every method used to update the class first gets a lock on the class. And while the locking code in the JVM has improved considerably since the 1.0 days, it is still not as highly performing as using a method that does not require a lock.

Now, using a thread-safe class is worthwhile only if the object is being simultaneously updated by more than one thread. However, this is rarely the case. Usually, the object is allocated and updated by a single thread. In that case it makes sense to use a class that does not have the locking overhead. The ArrayList, StringBuilder and HashMap classes (which were not available in Java 1.0) provide the same functionality as the prior classes, but because they are not thread safe they do not have the locking overhead.

#7: Use StringBuilder

Packaging data into a string is a common task in many applications. For example, a web page might have a login area at the upper right-hand corner of the page. If the user is not logged in, a login link appears. If the user is logged in, a hello message and a logout link appear. Here is one way to code such an area:

```
String login = "<div>";
if (loggedin) {
    login += "Welcome, ";
    login += user.getName();
    login += logoutLink;
} else {
    login += loginLink;
}
login += "</div>";
```

In the above code, the login string is built in pieces. The problem with the code is that strings are immutable. That is, they never change. The above code actually creates up to five string objects, only the last of which does not immediately become garbage.

When constructing a string in this fashion, it is better to use the StringBuilder class. The string builder class allocates a buffer into which the string can be built.

```
StringBuilder buf = new StringBuilder();
buf.append("<div>");
if (loggedin) {
    buf.append("Welcome, ")
        .append(user.getName())
        .append(logoutLink);
} else {
    buf.append(loginLink);
}
buf.append("</div>");
```

```
String login = buf.toString();
```

The end result is that only one object, the `StringBuilder`, needs to be garbage collected at the end of this code.

One more thought before you leave this topic. Consider this line of code:

```
String s = "Welcome, " + cust.getName();
```

Based on the previous text, you might think that this line would be better rewritten as:

```
String s = new StringBuilder(100)
    .append("Welcome, ")
    .append(cust.getName())
    .toString();
```

Actually, both lines perform the same. If you look at the compiled byte-code, it is the same for both lines. If you build a string in a single statement, the compiler automatically creates a temporary `StringBuilder` which it uses to build the string.

#6: Avoid `System.gc()`

The `System.gc()` method causes a major garbage collection. (The paper covers the differences between major and minor garbage collections later in item #2, Right-Size the Heap.) And because major collections are always costly, it is a good idea to avoid them if you can.

Some people advocate calling `System.gc()` during a quiescent state of the applications. Indeed, you will find that most application servers call `System.gc()` during initialization to flush out objects that were required during that initialization and will probably not be needed again.

While this advice is useful for a single-user desktop application, it is problematic within an application running within an application server. In such an application, there are hundreds, if not thousands, of concurrent users doing a variety of tasks and making a variety of requests. In such an environment, how could the application even begin to know when there is a quiescent state?

You can use the `-XX:+DisableExplicitGC` JVM argument to prevent the garbage collector from running in response to the `System.gc()` method.

#5: Right-Size Your Connection Pools

An application server manages two important connection pools – HTTP and database. You must allocate sufficient connections to handle the workload, but not so many connections that performance suffers.

This is one of those cases when more might be better but too much is worse.

Configuring HTTP Connections

If you are using Tomcat, or an application server that uses Tomcat as its servlet container, such as JBoss Application Server, then you can specify HTTP connection parameters in the `server.xml` configuration file. The properties related to HTTP connections are listed in table 1.

Table 1. Tomcat HTTP connection properties

Option	Default	Description
maxThreads	200	The maximum number of threads available to process requests. This value limits the number of requests that can be handled simultaneously.
minSpare Threads	4	The number of threads the web server tries to keep available above and beyond the number currently in use. The two spare threads settings are used to ensure that there are idle threads available to immediately handle future requests.
maxSpare Threads	50	If more threads than this are idle (not processing a request), then those threads are stopped and deallocated.
acceptCount	10	The maximum number of requests that can be queued, waiting for a thread to be freed. If the queue is full, the application server returns a 503 HTTP error.

Set `maxThreads` to handle the maximum number of simultaneous requests that the system can handle. The setting usually depends on the processing power of your hardware (faster processors and large memory allow for more simultaneous connections) and whether the processing is memory or disk bound. A lot of database access would be considered disk bound and enables a higher connection count because a portion of those connections will be waiting on a database response.

Set the `acceptCount` to the number of requests that you want to queue. If a typical request is handled quickly, then you can increase the `acceptCount`

without impacting the user experience too much. Most users will be more accepting of waiting an extra second or two to get a response than being given a 503 error.

The `minSpareThreads` and `maxSpareThreads` properties are listed in the Tomcat 5.5 documentation, but have disappeared in the 6.0 documentation. However, the code still stores these properties but it is unclear from a reading of the code if those properties are still used. Based on informal testing, it would appear that they are ignored.

WebLogic Server

WebLogic Server 9.0 and higher provides work managers that can automatically tune the number of threads available to run requests based on data kept on prior requests. You can influence the decisions made by a work manager by setting the `min-threads-constraint` and `max-threads-constraint` for that work manager in any of the XML configuration files (such as `config.xml` or `weblogic.xml`).

The work managers get their requests from a single, centralized queue. All TCP requests initially go into that queue. The `Accept Backlog` parameter defines the number of requests that can appear in that queue. If the queue is full, and additional requests get a 503 HTTP error.

WebSphere Application Server

In WebSphere, you can specify the read pool options using the Thread Pool page of the Web Container for your server within the administration console. Table 2 identifies the options you can set.

Table 2. WebSphere Web Container Thread Pool Options

Option	Default	Description
Maximum Size	50	The maximum number of threads available to process requests.
Minimum Size	10	The minimum number of threads available to requests.
Thread Inactivity Timeout	3500	The number of milliseconds a thread can be idle before it is deallocated.
Is Growable	False	Indicates if additional threads (beyond the maximum) can be allocated by the server if needed.

Configuring Database Connections

Each application server has its own mechanism for defining the number of database connections to maintain in the connection pool. But regardless of the mechanism used, there is one common caveat – ensure that the connections specified by the application server matches the number of connections allowed by the database. It does no good to set up a connection pool of 200 database connections if the database allows only 100 connections.

Table 3 describes the typical connection pool options provided by various application servers.

Table 3. Database Connection Pool Options

Gen. Option	Description
min connections	The minimum number of connections maintained to the database.
max connections	The maximum number of connections maintained to the database.
connection timeout	The amount of time a thread waits on a connection if all the connections are in use and the maximum connections have been allocated. If the time is exceeded, an exception is thrown.
idle timeout	The amount of time the application server waits before deallocating a connection that's no longer needed

The only difference between application servers is how you define each of the above options.

WebSphere Application Server

The administrative console has a Connection Pool page for every data source you declare. Table 4 lists, for each general option defined in table 2, the WebSphere-specific option, and its default value.

Table 4. WebSphere Application Server Database Connection Pool Properties

Gen. Option	Option	Default
min connections	Minimum Connections	1
max connections	Maximum Connections	10
connection timeout	Connection Timeout	180 seconds
idle timeout	Unused Timeout	1800 seconds

WebLogic Server

The Administrative Console has a Connection Pool tab for every data source you declare. Table 5 lists, for each general option defined in table 2, the WebLogic Server-specific option.

Table 5. WebLogic Server Database Connection Pool Properties

Gen. Option	Option
min connections	Initial Capacity
max connections	Maximum Capacity
connection timeout	Connection Reserve Timeout
idle timeout	Inactive Connection Timeout

JBoss Application Server

You use a `*-ds.xml` file to configure database connections. Table 6 lists, for each general option defined in table 2, the corresponding property in that file, and the default for the property.

Table 6. JBoss Application Server Database Connection Pool Properties

Gen. Option	Option	Default
min connections	min-pool-size	0
max connections	max-pool-size	20
connection timeout	blocking-timeout-millis	30 sec.
Idle timeout	idle-timeout-minutes	0 (never times out)

One quirk of the JBoss Application Server is that if a database connection is idle for `idle-timeout-minutes`, then the application server will drop that connection. If this causes the connection count to drop below `min-pool-size`, it will create a new connection. The end result is that, in an idle system, all database connections are dropped and re-established every `idle-timeout-minutes`.

#4: Prefer Newer JVMs

Every release of the JVM brings new performance improvements. So if you are still using an older JVM, consider upgrading to a newer one.

Recently, Intel developers have been working with the Sun JVM developers to improve the performance of JVM 6 [INTELJVM]. They have worked on things such as a tighter code generation and determining the best garbage collection algorithms to preserve locality of reference.

When a processor loads memory into its cache, it grabs more than just the one word being accessed; usually it grabs 64 or 128 bytes. If multiple pieces of data used by a given piece of code are located together, then they could be pulled into the cache at the same time thus improving the performance. This mechanism is called *locality of reference*.

Figure 1 illustrates the relative performance gains that have been made from JVM 1.3.1 through JVM 6 (1.6.0). The results were gathered by running an industry-standard benchmark with each on the JVMs on the same hardware and operating system.

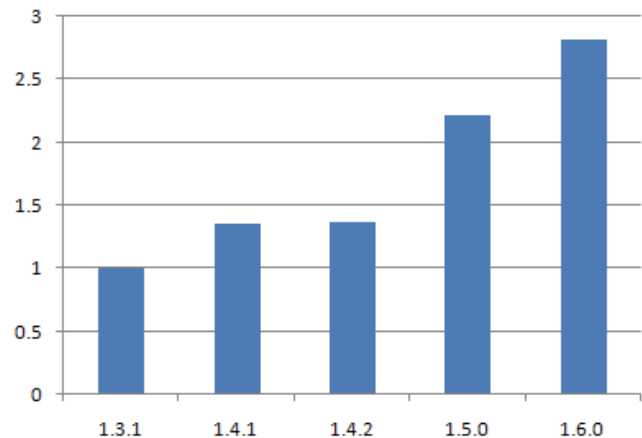


Figure 1. JVM Relative Performance Improvement

But the Sun JVM is not the only one that is improving. IBM continues to improve its JVM with each release. And likewise BEA (or actually, Oracle) continues to improve the performance of JRockit.

#3: Not All GC Algorithms Are Created Equal

One of the facts of life in a Java application is that objects die (become unreachable) and the memory used by those objects has to be recovered. Recovering that memory is the task of the garbage collector. The various JVMs provide different garbage collectors, each of which has their pros and cons. This paper focuses on the garbage collectors provided by the Sun JVM.

Understanding the Java Heap

It is essential to know how the Java heap is arranged to understand the various garbage collection

algorithms. The details that follow pertain to the heap as maintained by the Sun JVM; other virtual machines employ various alternate heap mechanisms.

As a Java application allocates objects, the virtual machine places those objects in a heap allocated by the virtual machine. When the application no longer needs the object by removing all references to that object, that object becomes unreachable. A garbage collection typically occurs when the virtual machine runs out of room to allocate new objects.

Most objects allocated by an application have a very short lifespan and are said to "die young." The heap is divided into the young generation, where new objects are placed, and a tenured generation, where objects that have survived several garbage collections are placed. Each generation is collected individually, and each has its own collection algorithms.

A simplified representation of the heap address space is shown in Figure 2. The young generation consists of the eden space, where all new objects are created, and two survivor spaces.

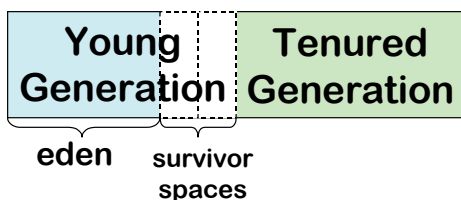


Figure 2. Java Heap

Additionally, there is a permanent space (not shown in the figure) which holds class objects, but that space typically does not participate in the collections and will not be addressed in this paper.

Understanding Garbage Collection

There are two categories of garbage collection: minor and major. A minor collection cleans out the young generation only, whereas a major collection cleans out the tenured generation and the young generation.

When the young generation is collected, all objects that are still in use are moved to one of the survivor spaces. After the collection, the eden space and the other survivor space are empty. If the survivor space fills up, any excess surviving objects are placed in the tenured generation. Additionally, any objects that have survived several minor collections are also placed into the tenured generation.

When the tenured generation is collected, all space occupied by unreachable objects is made available again. In most cases, the tenured generation is also compacted, with all surviving objects moved to one end of the tenured generation space.

When the new object allocator runs out of room in the eden space, the HotSpot Virtual Machine determines if a minor collection will suffice, or if a major collection is required. In the simplest algorithm, the virtual machine assumes that all objects will survive a collection and will thus overflow the survivor space and need to be placed into the tenured space. Therefore, if the tenured generation cannot hold all of the objects in the young generation, a major collection is required. For example, assume a heap size of 1000MB and a young generation size of 300MB. Once over 700MB of heap space is consumed (300MB for the young generation and over 400MB for the tenured generation), a major collection is performed. Note that if the heap is set to 1000MB, but it never fills beyond 700MB, there is 300MB of memory that the virtual machine is reserving but never using.

More sophisticated algorithms cause the virtual machine to first determine whether the tenured generation is sufficient to hold only the surviving objects from the young generation. For such an algorithm, full collections occur less often and better use is made of reserved memory.

Understanding the Serial Collector

The default garbage collector, also known as the serial collector, is used to clean both the young generation and tenured generation. For this collector, all application threads are stopped, a single thread performs the collection, and once the collection is complete the application threads are resumed. This collector compacts the tenured generation when doing a major collection.

Understanding the Parallel Collector

You can use these JVM arguments to run the parallel collector instead:

- `-XX:+UseParallelGC` – uses multiple threads to collect the young generation (introduced in 1.4.1). Starting with JVM 5, this collector is used by default if your system has at least 1GB of RAM and two CPUs.
- `-XX:+UseParallelOldGC` – uses multiple threads to collect the tenured generation (introduced in 1.5.0_06)

The parallel collector behaves similarly to the serial collector, but uses multiple threads to perform the collection. In general, it uses one thread per CPU (core), up to a maximum of eight.

Understanding the Concurrent Collector

The concurrent mark sweep (CMS) collector was developed to minimize the long pauses that can be

observed when doing a collection of the tenured generation by the serial collector. Provide the following JVM argument on the command line to run the concurrent collector:

- `-XX:+UseConcMarkSweepGC`

Expect to see newer garbage collectors in the future. JVM 7 will include a collector known as the garbage-first (G1) collector which is intended to replace the CMS collector and fix some of the issues with that collector [G1].

The CMS collector is run in a separate thread concurrently with the rest of the threads in the Java application. It performs the collection in phases, and stops the application threads only for those phases where it is absolutely necessary.

The first phase is an initial marking phase. During this phase, all objects directly reachable from a root reference are marked. Root references include those found in the stack, representing object references declared within methods, and those found in static variables. The application's threads are stopped for this phase, but this phase completes very quickly causing minimal disruption to the application.

The second phase is the concurrent marking phase. As the name implies, this phase is performed while the application threads are running. In this phase, the collector thread identifies the objects that are reachable by those objects marked in the first phase. Note that since the application threads are running that new objects could be allocated and existing references could be changed. To handle this situation, all new objects are marked. Additionally, objects containing references that are changed are tracked so that they can be reexamined.

The third phase is the final marking phase, also known as the remark phase. Once again, the application's threads are stopped. During this phase, the collection thread examines the references in the objects marked as changed since the initial marking phase. Once this phase is completed, all live objects are marked. In addition, some unreachable objects could be marked. For example, during the concurrent marking phase an object might have been reachable and marked as such, but later dereferenced by an application thread. Such an object would still be marked as live even after the remark phase and thus would not be collected. This object would, however, be collected during the next collection.

The fourth and final phase is the concurrent sweeping phase. The collection thread runs at the same time as the application threads and reclaims all of the heap

memory that was in use by objects that are not marked. Unlike the serial collector, which compacts the tenured generation, the concurrent collector instead maintains a list of available memory locations. To prevent fragmentation, it combines adjacent memory spaces into one larger space, and divides the heap into areas for allocation of objects of particular sizes. For example, all objects in the 10-20 byte size range might go into one area and all objects in the 100-150 byte range might go into another area. (These sizes are theoretical and should not be implied as actual sizes used by the Sun JVM.)

Concurrent collections have to be scheduled to start long before the tenured generation gets full to allow time for the collection. The first concurrent collection is started when the tenured generation is about 68% full. Later collections are started based on data, such as time to perform a collection, gathered during prior collections. Thus the efficiency of the collector improves the longer the application runs. Note also that several young generation collections (run by the serial collector) could take place during a single concurrent collection. By the way, you can run a parallel collector on the young generation using this option:

- `-XX:+UseParNewGC`

This option turns on a special parallel collector that behaves properly with the CMS collector. Do not use the standard parallel collector (`UseParallelGC`) with the CMS collector.

By default, the concurrent collector sets the survivor spaces to a very small size and the tenuring threshold to 0. Thus all objects surviving a young generation collection get tenured, increasing the frequency of tenured collections. It is done this way to prefer the concurrent collector. However, you can specify a survivor ratio and tenuring threshold on the command line to change this behavior:

- `-XX:SurvivorRatio=<number>`
- `-XX:MaxTenuringThreshold=<integer>`

The survivor ratio indicates how many times larger the eden space is compared to each survivor space. For example, a survivor ratio of 8 indicates that eden is 8 times as large as each survivor space. The ratio can be a decimal number. The tenuring threshold indicates how many times an object will be placed into a survivor space during minor collections before being promoted to the tenured generation.

Choosing a Collector

Which collector should you choose? That depends on your needs. If you need to maintain consistent

response times, use the CMS collector. If raw throughput is your overall goal and you can accept occasional long pauses, use the serial (single CPU) or parallel (multiple CPUs) collector. (This author would say that you should run a performance test with both and decide which works best for you, but that would be getting ahead of the countdown...)

#2: Right-Size the Heap

Even more important than picking the correct garbage collector is choosing the correct sizes for the heap and its generations. You can improve performance by as much as 35% just by changing the heap settings. The question is: how can you tell when you have the correct heap sizes? The answer is: gather heap usage data.

Gathering Heap Usage Data

The Sun JVM provides a number of command line arguments you can use to gather heap data. Here are some of them:

- `-verbose:gc` – prints overall heap usage before and after a collection, including the time it took to run the collection
- `-XX:PrintGCDetails` – prints usage of each generation before and after a collection
- `-XX:+PrintHeapAtGC` – prints detailed usage of each generation and its spaces' (eden, survivor spaces) usage before and after a collection.
- `-XX:+PrintGCTimeStamps` – prints the number of seconds the JVM has been running when the collection started.
- `-Xloggc:filename` – prints the collection data to the indicated file (otherwise prints to `stdout`).

Here is an example of the output when using `-verbose:gc`:

```
[GC 60176K->56377K
(65088K), 0.0050069 secs]
[GC 60473K->56675K
(65088K), 0.0049446 secs]
[Full GC 60771K->13067K
(65088K), 0.1216777 secs]
[GC 17163K->13507K
(65088K), 0.0038411 secs]
```

Note that the output identifies if the data is for a minor collection (GC) or major collection (Full GC). The data provided indicates (using numbers from first line as examples):

- the amount of heap space used before the collection (60176K)
- the amount of heap space in use after the collection (56377K)

- the size of the heap (65088K)
- the amount of time it took to perform the collection (0.0050069 secs)

Of course, looking at the data as text usually is not very helpful because it is difficult to spot trends. What you need to do is extract the data into a format that you can work with, such as a comma-separated-value (CSV) file, which you can load into a spreadsheet and graph. Most scripting language can do this for you. For example, here is a Windows PowerShell script that creates a CSV file from the `verbose:gc` data (enter it all on one line):

```
get-content $args |
  %{$_ -match "GC (?<before>[^K]+)K->
  (?<after>[^K]+)[^,]*, (?<time>[\\d\\.]*)" }
  | %{if ($_) {"" + $matches.before + "," +
  $matches.after + "," + $matches.time}}
```

Interpreting Garbage Collection Data

Loading the data into a spreadsheet such as Excel or OpenOffice.org Calc, you can generate a graph of the data. Figure 3 (next page) shows a typical plot of the data.

Several facts regarding the Java application can be asserted by examining figure 3. First, based on the X-axis, the garbage collector ran almost 2000 times.

Second, there were eight major garbage collections, and the program was heading towards a ninth major collection when it stopped. This fact is discernible using two different data points. First, knowing that a major collection takes significantly more time than a minor collection, observe that there are eight yellow dots (light triangles, when viewed in black-and-white) that do not fall into the almost straight line of dots barely above the X-axis.

Third, the dots for the heap space used before and after garbage collection rise until they come close to the maximum heap size, at which point there is a dramatic drop. It is this drop that signifies a major collection. Thus, the blue dot at the top of the rise, its corresponding magenta dot at the bottom of the next rise, and the corresponding yellow dot hovering way above the X-axis all correspond to the data for a major garbage collection.

Fourth, while the major collections free up large amounts of heap space (60MB in some cases), the minor collections free up very little. Based on the small difference in the height of the blue dots when compared to the magenta dots, it appears that minor collections free up about only two megabytes of heap space.

Adjusting Heap Size to Improve Performance

Based on the large number of minor garbage collections performed, and the small amount of memory recovered for each such collection, it follows that performance could be improved by allocating a larger young generation space. Ideally, with a larger young generation there would be no, or very few, major collections, only minor collections.

Using this knowledge the application was run again, this time specifying a young generation size of 20MB (about 1/3rd the size of the heap, which is 64MB). Graphing the `-verbose:gc` output from this run yields the graph shown in figure 4 (next page). The chart title provides the Java command line arguments used. The additional arguments, `-XX:MaxNewSize=20m` and `-XX:NewSize=20m`, set the young generation size to 20MB.

This chart shows about 1/3rd fewer garbage collections taking place, with each minor collection reclaiming about 17MB of heap space. There is also a decrease in the number of major collections.

There are a few other interesting points about this chart. First, notice that the minor collections are taking

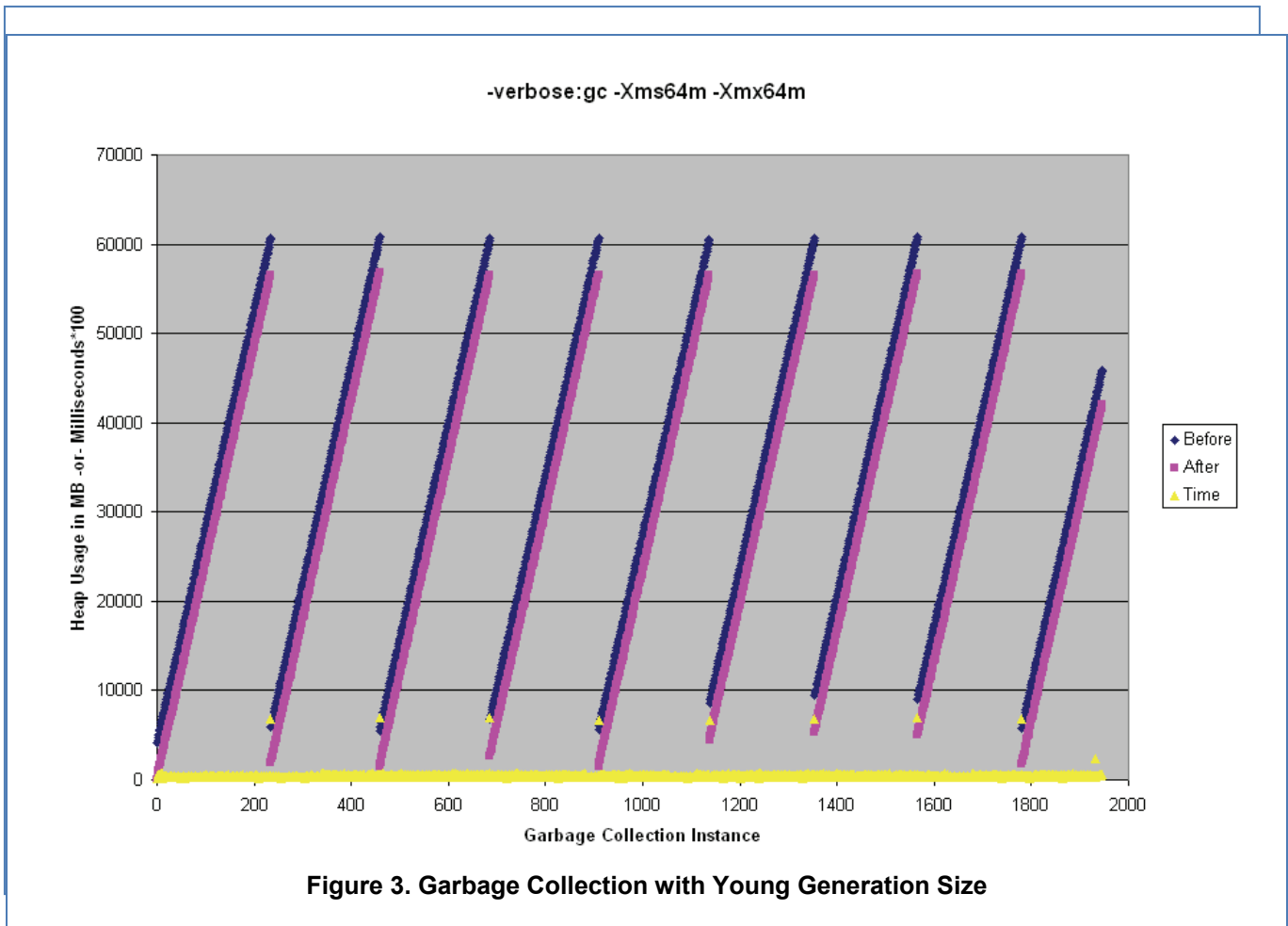
longer than on the first run. Examination of the raw data shows that minor collections are taking about two to three times as long as on the first run. This is to be expected since each minor collection now recovers more heap space (about eight times as much as in the first run).

Second, notice that the heap usage goes up to about 45MB before a major collection takes place, whereas on the first run, the heap usage went up to 60MB before a major collection. The JVM makes the pessimistic assumption that it might have to move all data from the young generation to the old generation. When acting on such an assumption would cause the old generation to run out of room, the JVM performs a major collection.

Though not ideal, the application achieved a 40% increase in throughput (transactions per second) by this one change.

Further performance improvement can be realized by adjusting the entire heap size, keeping the young generation size to 1/3rd the entire heap size. Figure 5, on the next page, shows the result of setting the heap to 128MB with a 40MB young generation.

Several facts can be gleaned from this chart. First,



only minor collection took place; there are no major collections. Second, heap usage has stabilized – with both the heap usage before collection (blue dots) and heap usage after collection (magenta dots) forming horizontal lines, one can assume that if the application ran forever that a major collection would probably never take place. This is an ideal situation. Third, the number of collections has been cut by almost half since the second run.

Compared to the first run, the performance (as measured by number of transactions performed per second) of the Java application increased by 67%, and the number of garbage collections were reduced by 80%. All of this was accomplished by gathering simple garbage collection statistics, graphing them so that the garbage collection pattern became obvious, and changing the command line parameters based on those patterns.

While the application used to illustrate these points is a small benchmarking program, these same techniques have been used to analyze real-world production applications. In one instance, analysis of the `-verbose:gc` output and proper application of heap size command line arguments reduced the time it took for an application to process a fixed number of transactions by 33%, thus meeting the customer's performance goal.

Recapping the Heap Settings

As a recap, here are the heap settings discussed in the above text:

- `-Xms` – the minimum heap size
- `-Xmx` – the maximum heap size. In a production environment, always set the min and max heap sizes to the same value.
- `-XX:NewSize` – the initial size of the young generation
- `-XX:MaxNewSize` – the maximum size of the young generation. In a production environment, always set the min and max young generation sizes to the same value.

#1: Testing, Testing, Testing

Given the audience for this page, the number one way to improve performance of a Java application should not come as a surprise. It is almost impossible to improve application performance if you are not aware of the bottlenecks that prevent the application from running optimally. And the best way to highlight such bottlenecks is via performance testing.

Ideally, such testing should be done in an environment that matches the production environment as closely as possible. In addition, the performance test should match typical user usage as much as possible.

If the bottleneck is in the Java application, here are some things you can try to eliminate or reduce the bottleneck:

- Adjust the heap sizes. See item #2 above for suggestions.

- Try another garbage collector. See item #3 above.
- Try a different JVM or later version of the JVM. See item #4 above.
- Check your connection pools. Especially look for excessive queued requests waiting for pooled resources. See item #5 above.
- Inspect the code looking for the performance inhibiting coding practices identified in items #6 through #10.
- Use a performance monitoring tool, such as JProbe, to pinpoint problem code.

Conclusion

As you can see, there are many things you can do to improve the performance of that slow Java application. This paper, and the papers referenced, should give you a good head start in finding and eliminating performance issues with your Java applications.

Glossary

CSV – Comma-Separated Value. A text file containing data, where individual pieces of data are delimited by commas.

CMS – Concurrent Mark Sweep. One of the garbage collector provided by the HotSpot JVM.

HTTP – Hypertext Transfer Protocol. The set of rules for transferring files on the World Wide Web

JIT – just-in-time compiler. A compiler technology whereby Java byte-codes are compiled into machine-specific opcodes at the time the Java application is run.

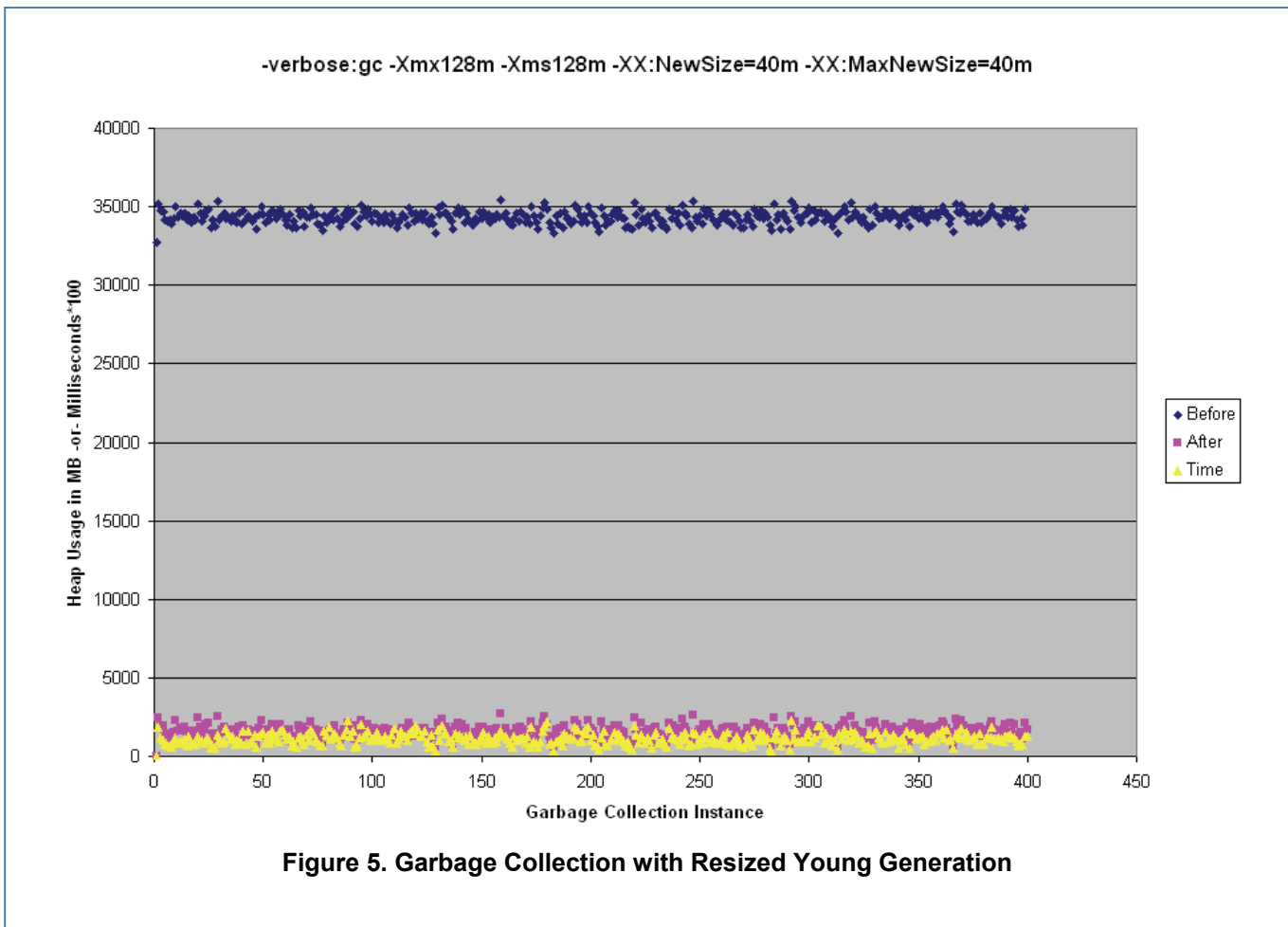
JVM – Virtual Machine for the Java Platform. The virtual machine used to run Java applications.

XML – Extensible Markup Language. An open standard for exchanging structured documents and data over the Internet.

References

[G1] Paul Ciciora, Antonios Printezis, “The Garbage-First Garbage Collector,” JavaOne 2008, San Francisco, CA
<http://java.sun.com/javaone/sf/sessions.jsp>

[GC131] “Tuning Garbage Collection with the 1.3.1 Java Virtual Machine”,



<http://java.sun.com/docs/hotspot/gc/>

[GC141] “Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1”,

<http://developers.sun.com/techtopics/mobility/midp/articles/garbagecollection2/>

[GC142] “Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine”,

<http://java.sun.com/docs/hotspot/gc1.4.2/>

[GCFAQ] “Frequently Asked Questions about Garbage Collection in the HotspotTM JavaTM Virtual Machine”,

<http://java.sun.com/docs/hotspot/gc1.4.2/faq.html>

[INTELJVM] Kingsum Chow, David Dagastine, Uma Srinivasan, “Maximizing Enterprise Java Performance on Multi-core Platforms”, JavaOne 2008, San Francisco, CA, May 2008,

[JVMOPT] “Java HotSpot VM Options”,

<http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp>

[TUNING1] “Java Tuning White Paper”,

<http://java.sun.com/performance/reference/whitepapers/tuning.html>

[TUNING2] “Tuning Java Virtual Machines (JVMs)”,

<http://edocs.bea.com/wls/docs61/perform/JVMTuning.html>

Copyrights and Trademarks

BEA, JRockit, and WebLogic are registered trademarks of BEA Systems, Inc.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

IBM and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries, or both.

All other brands and products referenced in this document are acknowledged to be the trademarks or registered trademarks of their respective holders.