Proceedings of the 5th WSEAS International Conference on Telecommunications and Informatics, Istanbul, Turkey, May 27-29, 2006 (pp51-56)

# Performance Implications of Memory Management in Java

DR AVERIL MEEHAN
Computing Department
Letterkenny Institute of Technology
Letterkenny
IRELAND
http://www.lyit.ie/staff/teaching/computing/meehan_avril.html

*Abstract:* - This paper first outlines the memory management problems that can arise in Java, and then proceeds to discuss ways of dealing with them. Memory issues can have serious performance implications particularly in distributed systems or in real time interactive applications, for example as used in computer games.

*Key-Words:* - Garbage collection, Java, Memory management.

## 1 Introduction

Problems with memory management range from slowing execution to system crashes, and are considered so serious that Java provides automatic garbage collection. While this removes some of the problems many remain [1]. Java memory management issues are especially relevant for developers who make significant use objects that reference large structures on heap memory or where rapid or consistent execution speeds are important. This applies in particular to the gaming industry, interactive real time applications, and to enterprise systems.[2]. Memory problems in Java are identified and approaches to solutions discussed.

A major advantage of object oriented programming languages (OOPL) is the information hiding that enables the programmer to work at a high level of abstraction facilitating development and code reuse. As objects are references to structures allocated in heap memory, programs developed in OOPL's can make extensive use of the heap.

While there is a body of work that concentrates on optimising compilation[3] or improving the garbage collector [4,5], only solutions applicable at the level of Java program code are considered here.

The next section outlines the kinds of memory problems that can occur in a programming language such as Java which has automatic garbage collection, while section three considers how Java programmers can best detect problematic sections of code, and section four discusses what can be done about it. It is important that any solutions do not negate the benefits of information hiding which were the impetus in developing OOPL's in the first place. The final section summarises the findings and makes recommendations.

## 2 Memory Management Problems

In the context of programming, garbage is any structure allocated in heap memory but no longer in use, i.e. there is no reference to it from within program code. Automatic GC removes the problem of dangling references where memory is marked as free for re-use while active pointers still reference it.

In Java the remaining problems are space leaks and slow or varying computation. Scalability is an important property of applications, but even if the program is scalable in all other aspects, its GC may not be scalable, and can cost a large percentage increase in execution time.

GC in Java is carried out on a low priority thread. This enables GC to make use of fee processor time, but it still has a cost and will pause the program for a short time. These costs can be significant in programs where objects are large in number or reference large structures on the heap [6]. For this reason full GC is only carried out when:

- explicitly called in code, or
- memory resources become critically low.

Java uses the Hotspot GC which divides the heap into generational sections, so that minor collections of the younger *Eden* space quickly reclaim structures references by short lived objects. Both the GC pause, and their frequency are potential problem areas.

Space leaks are also problematic, and occur when memory is no longer used, but has not been marked free for recycling. This can become excessive until memory is exhausted and the program crashes. Space leaks can occur in lots of ways. Using static objects inappropriately keeps them alive for the entire program. Problems occur when this object refers to a very large structure or to many others.

The seriousness of space leaks are relative to their size and the running time of the application. Small space leaks present few problems in short running applications that run on large memories. But even these can cause problems if the application is scaled up, or if it runs for a long time, e.g. on a server continuously, so that small leaks accumulate and eventually crash the system [7]. Even a few leaks are problematic if objects refer to large structures on the heap.

A more serious aspect is the difficulty in detecting space leaks as they often do not show up until testing. Fixing code late in development or worse, during maintenance, is well known for being costly. Unfortunately, being aware that a program has a space leak is only the first step. Identifying the section of code that caused the problem is problematic as the cause is not where the problem shows up [1, 7].

Even if the program does not crash, reducing available memory slows object allocation so that execution speeds can vary. Also there are likely to be more calls for GC, with a corresponding drain on efficiency.

The next section looks at how these problems can be identified and the following section considers what the programmer can do about these problems.

## 3 Identify Problematic code

The first step is to identify memory management problems, the earlier in development the better. Only then can something be done about it. Problem indicators range from system crashes to more subtle variable execution speeds over time.

The challenge is to monitor memory use while at the same time connect this to what is happening in the actual Java application code. It is too late when space leaks cause a crash as the program exits with the message:

```
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap
space
```

The challenge is to detect problems much earlier when something can be done.

### 3.1. Manual Inserts in Code

A global counter can be inserted into a class and updated in the Constuctor to indicate the number of objects of this class that have been allocated on the heap. In Java the *finalize()* method can be overridden to release resources such as sockets or file descriptors, and it is called just before garbage collection. Decreasing this counter in the classes' *finalize()* method means that the counter reflects the number of live objects, and can be checked at any point in the code. Alternatively, two counters can be used, one to record allocations, and one to record those that are available for GC. There is a snag with this however. The *finalize()* method leaves the object ready for GC, but there is no guarantee when, or even if, GC will occur.

It is important to monitor memory that is available for an application. This can be achieved using the *freeMemory()* and *totalMemory()* methods:

```
Runtime r =
Runtime.getRuntime();
float freeMemory = (float)
r.freeMemory();
float totalMemory = (float)
r.totalMemory();

System.out.println("Total
Memory = " + totalMemory);
System.out.println("Free
memory = " + freeMemory );
System.out.println(((totalMemo
ry - freeMemory)/1024) + "K
used");
```

This has the following output when run on a windows OS:

```
Total Memory = 2031616.0
Free memory = 1868432.0
159.35938K used
```

These extra lines of monitoring code are relatively simple to do. They could be put into a method that dumps the information, with appropriate comments to identify where the method was called from, to a file for later processing.

This is useful for giving an overview of the whole of a program, for example comparing memory use of objects at the beginning, at key points and also at the end of program execution. But the information is limited, omitting useful details such as the GC algorithm used, or how the different heap spaces are utilised.

Fortunately it is also possible to monitor garbage collection and memory usage in other ways.

### 3.2. Using Command Line Arguments

The garbage collection command line argument *verbosegc* can be used to diagnose problems with garbage collection. The output gives information on the garbage collector as well as occupancy of the young generation, and of the entire heap before and after a collection as well as the length of pause while GC was carried out. At the command line:

```
java
    -verbosegc
    -XX:+PrintGCDetials
    <CLASSFILENAME>
```

or using a jar file:

```
java
    -verbosegc
    -XX:+PrintGCDetials
    –jar <JARFILENAME.jar>
```

This results in output of the format:

```
[GC [<collector>: <starting
occupancy1> -> <ending
occupancy1>, <pause time1>
secs] <starting occupancy3>
-> <ending occupancy3>,
<pause time3> secs]
```

Unfortunately this generates a lot of data. For a small test program that ran 8 seconds, 19 pages of statistics were generated. Wading through this would reveal what was happening at every instance of the program, but unless print statements are inserted at specific points in the code (these will be output along with the GC statistics) matching the statistics to specific sections of code is difficult.

### 3.3. Java Monitoring and Management API

From Java 1.5, the Java Monitoring and Management API provides a way to monitor memory usage and GC statistics. There is much more to this API, but only these aspects are considered here. This includes the monitoring programs jstat and JConsole [8] as well as Mbeans. These are briefly outlined here.

The *jstat* command prints statistics on garbage collection, but is not available on all platforms. (It was previously jvmstat ) It is used on the command line with the identifier for the executing JVM, for example:

```
jstat -gc <jvmid>
```

(Gives information on how the various sections of the heap are used as well as the number and times of GC.)

The identifier of the JVM can be obtained by using *jps* at the command line. The problem here is the program has to be running to obtain its identifier, which can cause difficulty with short running programs. Inserting a temporary pause or a infinite loop will keep the program alive for this testing purpose if necessary. Results can be appended to a text file to make it easier to use them later.

```
C:\mmtest>jstat –gcutil 1608
                    >> gcresults.txt
```

Among several alternative parameters for *jsat* include *-gccapacity* (for statistics about capacities of different regions in heap memory) *-gcutil* (for a summary of GC statistics). The information available with this tool is very detailed, including the number and extent of GC; the usage of different generational sections of the heap; time taken etc.

*jstat* gives information at the end of execution, but it is often useful to know what is happening every time GC is called (as shown for example using the *-verbosegc* option).

*JConsole* is a GUI tool that sits on top of the new Monitoring and Management API, providing information on memory use, garbage collections, threads, classes, and Mbeans. JConsole uses JMX to monitor the JVM and can be used either locally or remotely. To use it on the localhost

```
java
–Dcom.sun.managementjmxremote
<PROGRAM>
```

starts the *PlatformMBeanServer*. *JConsole* graphically displays the various MemoryPools of Eden, Perm Gen, Survivor and Tenured Gen. This tool is useful for detecting low memory, enabling and disabling GC, using verbose tracing, and using Mbeans [9, 8].

The Java management Extension (JMX) API is now integrated with J2SE. GC monitoring is achieved using *Mbeans*, java objects which represent resources. When an *Mbean* is registered in a *Mbean* Server it becomes an agent.

Sun's java.lang.management contains many useful classes that can be used to monitor memory levels. For example the MemoryMXBean includes the interface

```
javax.management.NotificationE
mitter
```

which can be used for notifications of the type:

```
MEMORY_THRESHOLD_EXCEEDED
```

which can be implemented on a thread. Other useful *Mbeans* for monitoring GC are:

```
GarbageCollectorMXBean,
MemoryManagerMXBean,
MemoryPoolMXBean
MemoryMXBean
```

For example:

```
public interface
GarbageCollectorMXBean extends
MemoryManagemnetMXBean
```

This interface can be used to detect the total number of collections ( *getCollectionCount()* ) and the time this took (*getCollectionTime()* )

MBeans can be used to send a warning message when memory levels become low. When this was tried, it worked well in some test programs, but in other tests the warning message often did not occur, or if it did it was too late to prevent a crash.

### 3.4. Memory Profile Tools

Memory profile tool such as JProbe Memory Debugger or GCspy provide graphical output that represents the usage of the heap as a program executes. Such tools slow down execution too much to be used in a final product but they are useful during development to highlight problem areas.

Visual tools graphically chart heap use as peaks and troughs as objects are created and destroyed. Usually a program will increase memory use until it reaches a peak, then oscillate between peaks and troughs that remain relative constant. If the peak level of heap use increases steadily the application has space leaks because it is keeping structures referred to by objects. This could be the nature of the program or it could be that objects are not released so that the garbage collector can free them.

Another danger sign is frequent occurrences of GC which do not significantly increase the level of available heap, and/or GC which takes a long time.

For example in Quest's JProbe break points in the code are indicated as lines on the graph making it easier to connect the display with the code itself. This tool indicates what objects are live at whatever point in the execution is being studied. Sometimes an object has been deleted in the code but is still referenced by another variable. If this is the problem, then JProbe will show that object is still there. Using JProbe's Reference Graph it is possible to identify all objects that reference this object. It is then a matter of trawling through these and looking for the code that references it.

Another example is Visual GC which does not come with JSE as standard on all platforms, but for JSE 5 download jvmstat 3.0 which includes it. Although VisualGC is a very useful tool, it's use is not straightforward. If the developer is using an IDE it is necessary to begin execution of the program, switch out of the IDE to the command window, run the *jps* command to get the *vmid*, which is then uses to run visualGC. This is a problem that may soon be sorted, for example Eclipse already allows VisualGC to be used within it and downloading Milestone 2 enables Visual GC integration in Netbeans (only version 4.0 or newer releases).

## 4 Finding Solutions

Once the information on memory usage and GC has been gathered, and the problems identified it is then possible to use this to rectify the problems and to improve efficiency. This sections considers the various approaches to solving GC problems and assesses them in terms of their ease of use, their effectiveness and how well they keep the information hiding which is the motivation for using OOPL in the first place

These solutions are either implemented by making amendments at the level of the code, or by fine tuning GC itself.

### 4.1 Program Code

Once the problematic areas of code are identified it may be possible to do something at the code level. For objects that remain live after they are no longer in use, simply setting them to null may not always solve things, especially if the structures they reference are very large [10].

While it is important to avoid a system crash with the *OutOfMemoryError*, the information gathered on memory levels described in the previous section can be used to make sure a program never runs very low on available memory. To call the garbage collector explicitly:

```
Runtime r =
Runtime.getRuntime();
r.gc();
```

Davis [11]suggests object pooling where a generic abstract class is used to handle storage and tracking while the concrete subclasses deal with instantiation,

validation an destruction as needed. This approach resulted in an 88% reduction in execution time in a real time application.

Another approach at the code level is to use abstract class *Reference* and subclasses, SoftReference, WeakReference and PhantomReference. Garbage collection can occur even if these references refer to an object. A *ReferenceQueue* can be used to register soft, weak or phantom reference objects (in the case of phantom references it must be used). These can be used for monitoring and also for control of the order that references are freed[12, 13].

The disadvantages of modifications to program code include an increase in complexity and loss of information hiding.

### 4.2. Use Adaptive GC Ergonomics

It is possible to manually fine tune GC. Command line options and environment variables can be used to alter the memory management behaviour of Java's Hotspot GC on both Client virtual machines (VM) and Server VM's to modify defaults such as the heap size, whether parallel GC is enabled, upper time limit for GC, time ratio of GC to application time, etc. Often an advantage in one area will be at a cost in another, hence any decisions need to take an overall view.

For example, the size of the heap has an effect on the frequency and timing of GC. It can also have a bearing on fragmentation. If the heap is too small there can be fragmentation problems, and more frequent GC. On the other hand if the heap is too large, GC may not occur as often but when it does, the time for collection can be excessive. The choice of heap size needs to consider peak loads to be able to cope with these. The aim is to reduce frequency of GC by considering:

- the size of young and old generations - increasing young generations will reduce the frequency of GC but will increase the duration of pause
- the load on the heap – the faster the heap fills, the greater the frequency of GC
- the life-time of objects – live objects take up space, so an increase in lifetime increases the frequency of GC. Best to keep lifetime to a minimum

It is possible to fine tune GC with respect to young generation, old generation and survivor space sizes as well as the tenuring threshold. Increasing the size or the heap and/or increase the upper time limit allowed for GC. Setting command line flags can be used to achieve this, e.g.

```
-XX:MaxGCPauseMillis=n
-XX:GCTimeRatio=n
```

Unfortunately it is not always clear which adjustments are the most beneficial, and what works for one program may not be the best approach for another. It is often a matter of priorities, deciding what is important for a particular program. It may be necessary to experiment

The default choice of garbage collector is often fine, especially for uniprocessor or relatively small programs, but may not necessarily be best in programs that use a large number of threads, processors and/or sockets, or which involve a large amount of allocated memory on the heap.

Java now allowa the programmer to select a garbage collector to suits a particular application. This is important because Java applications range from desktop programs that run on a single machine to distributed systems, so the use of allocated memory on the heap can be very varied.

In order to make an informed choices about GC it is important to understand how the various approaches to garbage collection operate and how allocated heap memory is organised in Java. For a particular program, the programmer needs to consider the way that allocated memory is used when that program is executed.

If the pauses for GC of the young space is too long, using the parallel young generation collector can be helpful in reducing them. This is selected using the following command option with verbosegc:

```
-XX:+UseParallelGC
```

or

```
-XX:+UseParNewGC
```

If the pauses for major GC are too long, using the concurrent mark sweep low pause collector can help:

```
-XX:+UseConcMarkSweepGC
```

The use of adaptive GC ergonomics can have a significant effect, yet does not interfere with the development process. This has the major advantage that the developer can work at a higher level of abstraction, which is the motivation for using an OO language [14].

## 5 Conclusion

Even if memory problems are not suspected it is useful to check for them. Testing should include running programs for long periods of time to highlight small space leaks that could potentially become a problem. While Java's new memory and management API is valuable here it is in the provision of JConsol that best eases the process of performance tuning GC. Using a graphical profiling tool is the easiest way to monitor memory usage and yields faster results than manually inserting debugging code or dumping GC information to the screen or to a file. Other commercially available graphical memory profiling tools are equally useful, but may require leaning a new skill-set.

Once the problems have been identified, it is important to work on a solution. This is where tools are less useful, and programmer knowledge essential. Any solution will need to be cognisant of how the Hotspot GC works, what the memory management problems reveal about what is happening in the memory as the code runs, and the requirements of the particular development project. Often there will not be any one clear solution, but compromises will be needed. Such judgements are beyond presently available tools, which are most useful for providing information on memory use, and for indicating success or otherwise of changes that might be tried out.

Hard coding memory management code into a system detracts from the abstraction and information hiding which make object oriented programming languages so easy to use. It is best if a solution can be found that operates independently of the code.

### *References:*


[1] A.Meehan, Java Garbage Collection – A Generic Solution?, *Information and Software Technology* Vol.43, 2001, pp. 151-155

[2] A Pankajakshan, Plug Memory Leaks in Enterprise Java Applications, 2006 www.javaworld.com/javaworlk.jw-0302006/jw-0313-leak_p.html

[3]T. Suganuma, T.Yasue, and T.Nakatani, A Region-based Compilation Technique for Dynamic Compilers, *ACM Transactions on Programming Languages and Systems*, Vol. 28, Issue 1, pp. 134-174, January 2006.

[4] Y. Levanoni, E. Petrank, An On-the-fly Reference-counting Garbage Collector for Java, *ACM Transactions on Programming Languages and Systems*, Volume 28 , Issue 1, 2006.

[5] D.Bacon, D.Attanasio, H.Lee, S.Smith, *Java without the coffee breaks: A nonintrusive multiprocessor garbage collector*, Proceedings of the SIGPLAN 2001 Conference on Programming Languages Design and Implementation. ACM Press, pp 92-103, 2001

[6]D Sosnosk, Smart object-management, 1999 http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance_p.htm

[7] J. Patrick, Handling Memory Leaks in Java Programs, 2001 www.128.ibm.com/developerworks/java/library/j-leaks/

[8] R. Donepudi, Use a Profiler to Make Your Java Apps JVM-Friendly, 2005 *www.devx.com/Java/Article/30264*

[9] M. Chung, Using Jconsole to Monitor Applications, 2004 java.sun.com/developer/technicalArticles/J2SE/jconsole.html

[10] J. Shirazi, *Nulling* Variables and Garbage Collection, 2002 www.javaspecialists.co.za/archive/Issue060.html

[11] T. Davis, Improve the robustness and performance of your ObjectPool, http://www.javaworld.com/javaworld/jw-08-1998/jw-08-object-pool.html

[12]M. Pawlan *Reference Objects and Garbage Collection*, htpp://developer.java.sun.com, Aug.1998.

[13] A Dãncus, Garbage Collection For Java Distributed Objects, MSc Thesis, Worcester Polytechnic Institute, 2001.

[14] A Meehan, Garbage Collection and Data Abstraction Based Modular Programming, PhD thesis, University of Ulster, 1999.