By Performance Metrics Analysis of CLR & JVM– A Survey

Mrs. Pratibha Singh, Mr.Dipesh Sharma, Mr.Sonu Agrawal

RIT, Raipur, Dept.of Computer Sci. & Tech.dept.of information & tech.

ABSTRACT

This paper is mainly designed to show the evidence to prove difference between the Java Virtual Machine (JVM) and Common Language Runtime (CLR). The performance is measured with Execution time, Memory Management and Garbage Collection while executing the programs.

General Terms

Java Virtual Machine, Common Language Runtime.

Keywords

Execution time, Memory Allocations and Garbage Collection.

I. INTRODUCTION

Java Virtual Machine (JVM) was introduced by Sun Micro System and Common Language Runtime (CLR) was introduced by Microsoft for their Java and .Net respectively. Though these two have similar architectures the differences occurs on their performances. In this paper we implemented the performance comparison of Microsoft's CLR and Sun's JVM on the Windows platform.

[All the experiments in this paper done on the standard personal computer of configuration 40GB hard disk, 1GB Ram, Intel processor of 1.88 GHZ Speed and the Software configuration of JDK1.6 and Microsoft visual studio 2010.]

1. *Java Virtual Machine (JVM):*

Java is an object oriented language. Java codes are converted into Java byte code with the help of Java compliers. Java class files contain these Java byte codes and these are independent of different platforms. JVM is used to handle that class file at run time. Class file is not directly run on the host machine it needs to be converted to the host machine's language. This conversion is done by the JVM. JVM has the ability to handle automatic memory management. It performs Noncontiguous memory allocation.

2. *Common Language Runtime (CLR):*

In .Net the source code written in the languages such as C# or VB.NET. At compile time, .Net compiler converts source code into CIL code or Common Intermediate Language code (also called as MSIL—Microsoft Intermediate Language) which is in the form of byte code. At run time just-in-time (JIT) compiler converts the CIL code into native code to the operating system.

The .NET Common Language Runtime (CLR) is designed to be a language-neutral architecture. It handles memory allocation, error trapping, and interacting with the operating-system. It has characteristics of JVM like automatic memory management. But it performs contiguous memory allocation.

3. *Execution Time:*

Byte code commonly known as class files in Java and executable files in .Net. Size of the byte code depends on the complier. Generally smaller in byte code leads to quick transmission of code on the networks.

Performance also lies on execution time of programs. But execution time not depends on size of the code. It is because sometimes larger programs run in quicker time and smaller programs takes longer time to execute.

Let us consider an example for the performance of CLR and JVM. The execution time of similar programs is calculated.

In Java,

```
public class Performance{
public static void main(String args[]){
long startTime;
long endTime;
int x;
startTime=System.currentTimeMillis();
for(int i=1;i<10000000;i++){
x=i*234;}
endTime=System.currentTimeMillis();
System.out.print("Execution Time:");
System.out.println(endTime-startTime);
}}
```

The output is,

```
C:\JVMvsCLR>java Performance
Execution Time:24

C:\JVMvsCLR>java Performance
Execution Time:25

C:\JVMvsCLR>java Performance
Execution Time:26

C:\JVMvsCLR>java Performance
Execution Time:24

C:\JVMvsCLR>java Performance
Execution Time:24

C:\JVMvsCLR>java Performance
Execution Time:23

C:\JVMvsCLR>java Performance
Execution Time:24
```

In C#,

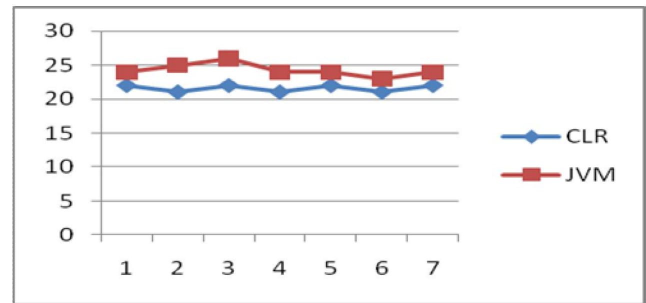
```
using System;
static class Program{
static void Main(){
long startTime;
long endTime;
int x;
startTime=DateTime.Now.Millisecond;
for(int i=1;i<10000000;i++){
x=i*234;}
endTime = DateTime.Now.Millisecond;
Console.WriteLine("Execution Time:" + (endTime -
startTime).ToString());}}
```

The output is,

```
c:\JVMvsCLR>Performance
Execution Time:22

c:\JVMvsCLR>Performance
Execution Time:21

c:\JVMvsCLR>Performance
Execution Time:22

c:\JVMvsCLR>Performance
Execution Time:21

c:\JVMvsCLR>Performance
Execution Time:22

c:\JVMvsCLR>Performance
Execution Time:21

c:\JVMvsCLR>Performance
Execution Time:22
```

The following graph explains the things better to understand. The attributes of the graph taken from values of the output at seven different times.

The Performance Graph,



[Note: Smaller in value, better in efficiency]

According to the performance in the execution time of similar programs as shown, The CLR performance was always better than the JVM's performance.

4. *Memory Allocations:*

Both the CLR and the JVM manage an internal heap of memory that is used for allocations (**heap**-a memory area used by the JVM and CLR for Dynamic Memory Allocation).

In JVM, it places a fixed upper limit on the heap size (by default 64Mb). If the JVM tries to satisfy an allocation that would result in the heap growing beyond that limit, and no garbage can be collected, then an OutOfMemoryError is thrown and the allocation fails.

But in CLR, it has no such artificial upper limit on the heap size. The CLR heap maximum size will be dependent on how much memory can be allocated from the operating system.

A similar program shows how the memory allotted to objects and other things in JVM and CLR.

In JVM,

```
import java.util.*;
public class OOM {
public static void main(String args[]) throws Exception{
long total=500000;
long used=0;
List<byte []> l=new ArrayList<byte[]>();
long times=100000;
try{
for(int i=1;i<total;i++){
byte []t=new byte[1024];
l.add(t);
if(i %times==0)
System.out.println("AllocatedMemory:"+getMemoryUsed());
}
System.out.println("Memory Allocated");}
catch(OutOfMemoryError e){
System.out.println(e);}}
public static void getMemoryInfo(){{
Runtime runtime=Runtime.getRuntime();}}
```

```
public static long getMemoryUsed(){Runtime
runtime=Runtime.getRuntime();
Long used=runtime.totalMemory()-runtime.freeMemory();
return used;}}
```

The Output,

```
c:\JVMvsCLR>java OOM
Allocated Memory:105034432
Allocated Memory:210245448
java.lang.OutOfMemoryError: Java heap space
```

In C#,

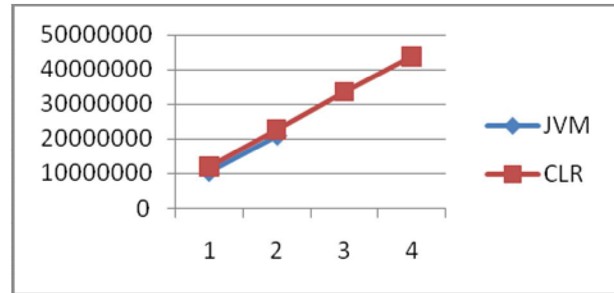
```
using System;
using System.Collections.Generic;
using System.Diagnostics;
namespace ConsoleApplication{
class Program{
static void Main(string[] args){
long total = 500000;
long times=100000;
long used;
List<byte[]> l = new List<byte[]>();
try{
for (int i = 1; i < total; i++){
byte[] t = new byte[1024];
l.Add(t);
if(i %times==0)
Console.WriteLine
("Allocated Memory:"+GetMemoryUsed());}
Console.WriteLine("Memory Allocated");}
catch (OutOfMemoryException ex) {
Console.WriteLine(ex);}}
public static long GetMemoryUsed(){
Process process=Process.GetCurrentProcess();
return process.PrivateMemorySize64;
}}}
```

The Output,

```
c:\JVMvsCLR>OOM.exe
Allocated Memory:119087104
Allocated Memory:228007936
Allocated Memory:335511552
Allocated Memory:438460416
Memory Allocated
```

According to the output due to the upper limit condition JVM fails to allocate but CLR doesn't. The performance Graph clearly shows how JVM lagging to allot the memory and CLR memory allocation.

The Performance Graph,



In memory allocations also CLR performance will be higher than the JVM's performance.

5. *MEMORY RELEASING:*

There are two types of memories used while executing the programs. Heap memory and Operating System memory. Heap memory stores all the objects created by executing a program. Objects which are created by **new** operator and memory for new objects are allocated on the heap at run time. Operating System memory is used to store the programs and Execution process details in it.

5.1 Garbage Collection:

Garbage collection is the process of automatically freeing objects and its memory space in the **Heap** which are no longer needed by the program. When an object is no longer referenced by the program, the heap space it occupies must be recycled so that the space is available for subsequent new objects. The garbage collector determines which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects.

The following Programs illustrated how Garbage Collector recovers the Heap Memory which is no longer needed by the program,

In CLR,

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
namespace ConsoleApplication{
class Program{
static void Main(string[] args){
List<String[]> list=new List<String[]>();
double before,after;
String []t;
for(int i=0;i<10;i++){
t=new String[10000];
list.Add(t);}
Console.WriteLine();
before=ByteToMB(GC.GetTotalMemory(false));
Console.WriteLine("MemoryUsed[KB]:"+
before.ToString("0"));
list=null;
GC.Collect();
after=ByteToMB( GC.GetTotalMemory(true));
```

```
Console.WriteLine("After GC");
Console.WriteLine("Memory
Used[KB]:"+after.ToString("0"));
double pert=((before-after)/before)*100;
Console.WriteLine("Percentage
ofDeAllocation:"+pert.ToString("0.00"));
Console.WriteLine();}
public static long ByteToMB(long inBytes){
return inBytes/ 1024;}}}
```

The output,

```
Memory Used[KB]:437
After GC
Memory Used[KB]:41
Percentage of De Allocation:90.62
```

Initially the heap as the memory of 437KB, after Garbage collection works finishes it memory pull down to 41KB. Nearly it deallocated 91% of memory.

In JVM,

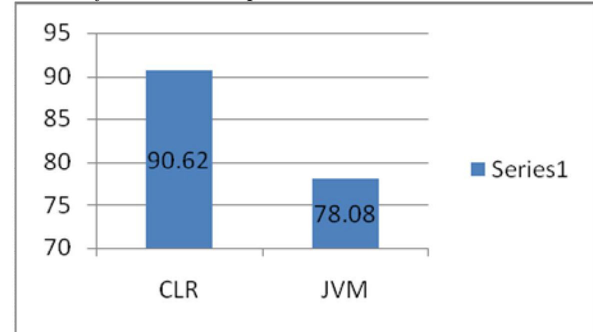
```
import java.util.*;
public class JVMGC{
public static void main(String args[]) throws Exception{
double before,after;
List<String[]> list=new ArrayList<String[]>();
String []t;
for(int i=0;i<10;i++){
t=new String[10000];
list.add(t);}
before=ByteToMB(getUsedMemory());
System.out.printf("\nMemory Used [KB]:%.0f",before);
list=null;
System.gc();
after=ByteToMB(getUsedMemory());
System.out.print("\nAfter GC");
System.out.printf("\nMemory Used[KB]:%.0f",after);
double pert=((before-after)/before)*100;
System.out.printf("\nPercentage of De Allocation:
%.2f",pert);
System.out.println();}
public static long getUsedMemory(){
Runtime runtime=Runtime.getRuntime();
return runtime.totalMemory()-runtime.freeMemory();}
public static long ByteToMB(long inBytes){
return inBytes/1024;}}
```

The output,

```
Memory Used [KB]:583
After GC
Memory Used[KB]:128
Percentage of De Allocation: 78.04
```

Initially the heap as the memory of 583KB, after Garbage collection works finishes it memory pull down to 128KB. Nearly it deallocated 78% of memory. This is considerably less than the CLR performance.

The Performance Graph,



Here also the performance of CLR is considerably higher than the performance of JVM.

5.2 Releasing OS Memory:

Apart from the Heap memory other important memory which is used to execute the program is operating system memory. In JVM, in Garbage Collection it Releases only the memory of heap but it release the memory of operating system. In fact it never releases the operating system memory even though it no longer needed. On the other hand, In CLR it will release allocated memory back to the operating system if it is no longer needed.

6. CONCLUSION:

In this paper we experiment the things like execution time, memory management and Garbage collection in Windows Platform. Though both JVM and CLR have these features but on the performance wise CLR is better on all the occasions then the JVM.

ACKNOWLEDGMENTS

Our thanks to the MRS. UZMA MAM for his valuable contribution towards this paper.

REFERENCES

[1] **Sam Shiel** and **Ian Bayley,** March 5, 2005. A Translation-Facilitated Comparison between the Common Language Runtime and the Java Virtual Machine.

[2] **Jeremy Singer**, University of Cambridge Computer Laboratory. JVM versus CLR: A Comparative Study.

[3] **Herbert schildt** 2002.Java 2, Fifth edition. McGraw Hill publications.

[4] **Bruce Eckel** 2000.Thinking in Java, Second Edition. Prentice Hall, New Jersey.

[5] Matthew MacDonald and Mario Szpuszta, Pro ASP.NET in C# 2005, Apress 2005.

[6] **Robin A. Reynolds-Haertle,**OOP with Microsoft Visual Basic .NET and Microsoft Visual C# Step by Step by Microsoft Press 2002.