DYNAMIC MEMORY ALLOCATION – CLR PROFILER

Adrian LUPAȘC
alupasc@ugal.ro
Viorica POPOIU
vyo.rash@yahoo.com
Dunarea de Jos University of Galati, Romania

At the present time, information systems are an important component in the development of many activities, due to their ability of managing a large amount of data and performing complex operations in a very short time. In this regard, very important is the correct management of all available resources, especially memory management. Haven't happened to test an application, but it runs out of memory? This paper is trying to explain some of the situations causing those problems and also is trying to find some solutions for fixing them. Concluding, the purpose of this paper is to emphasize on one hand the importance of using a memory profiler in the development of your applications and on the other hand, the advantages brought by CLR Profiler on your .NET applications.

Keywords: memory profiler, garbage collector, heap memory, .NET Framework, String, StringBuilder, CLR Profiler

JEL Code: C88, D83, L86

1. Introduction

At the present time, no matter what domain of activity is taken into consideration, information systems are considered to be essentials, because they significantly reduce the effort associated with the human factor of an entity, and also improves and simplifies considerably activity flows. But if we want to get maximum benefit from an information system, it is important to efficiently use every available resource at the computer system level. Therefore, the design and implementation of an information system should be done by taking into account all aspects that depend on the efficient allocation of computer system resources. In this regard, this paper aims to present the aspects related to the efficient management of computer memory, through information systems developed on .NET Framework, using a tool provided by Microsoft, called *CLR Profiler*.

At the operating system and the applications level, a strong impact on performance, has the way in which the memory allocation is made. Also, the memory allocator is always a critical component, because any implementation mistake may seriously affect the proper functioning of programs. This danger is amplified by the fact that the proper management of the free and occupied areas is generally a complex and difficult process, where errors can occur relatively easy.

Therefore, the main objectives considered in this paper have been related to:

- presentation of CLR Profiler tool and its functionalities, the facilities provided by it and the analysis made by covering the main differences between the String and StringBuilder classes;
- presentation of the way in which the dynamic allocation of memory is made;
- the functioning mode of garbage collector.

In this context, this paper is divided into several parts: an introductory part in which are briefly summarized relevant information concerning CLR Profiler, and also, a succinct description of its functionalities. In the second part of the paper is described the manner through which the dynamic allocation of memory is made, after which is presented the functioning mode of *garbage collector* component. Then, is presented the dynamic allocation of memory at the level of an application made in Visual C# 2010 Express, presentation being accompanied by some screenshots, in order to facilitate the proper understanding of the main functional characteristics of CLR Profiler. In the last part we have presented some conclusions and the main bibliographic sources that formed the basis for this paper.

2. What is CLR Profiler?

CLR Profiler is a tool that can be used with .NET Framework, being offered by Microsoft. It allows the user to investigate in real time the allocation of heap memory and garbage collector's behavior too, according to a profiled application.

Using CLR Profiler, the programmer is able to easily identify the code sequences that cause an inefficient allocation of memory and, which also determines the existence of too many objects that have to be collected by the garbage collector. As we know, the memory is an important resource and its efficient allocation can guarantee the proper functioning of the future developed information system, but also benefits and optimal results, especially in terms of time response for various processing.

Thus, through this tool, we can see in real time how much memory has been allocated during the execution of an application, but we can also view the objects created when methods are called.

This tool provides different ways of viewing how memory allocation is done and garbage collector's behavior, according to MSDN[25] as follows:

- Histogram Allocated Types – gives a high-level view of what object types are allocated (by allocation size) during the lifetime of your application. This view allows you to click parts of the graph so that you can see which methods allocated which objects;
- Histogram Relocated Types – displays the objects that the garbage collector has moved because they have survived a garbage collection;
- Objects By Address – provides a picture of what is on the managed heap at a given time;
- Histogram By Age – allows you to see the lifetime of the objects on the managed heap;
- Allocation Graph – graphically displays the call stack for how objects were allocated;
- Assembly, Module, Function, and Class Graph – these four views are very similar. They allow you to see which methods pulled in which assemblies, functions, modules, or classes;
- Heap Graph – shows you all of the objects in the managed heap, along with their connections;
- Call Graph – lets you see which methods call which other methods and how frequently;
- Time Line – displays what the garbage collector does over the lifetime of the application. Use this view to investigate the behavior of the garbage collector, determine how many garbage collections occur at the three generations and how frequently they occur and determine which objects survive garbage collection and are promoted to the next generation;
- Call Tree View – provides a text-based, chronological, hierarchical view of your application's execution.

System requirements (CLR Profiler for .NET Framework 4):

- Windows 7, Windows Server 2003, Windows Server 2008, Windows Server 2008 R2, Windows Vista, Windows XP;
- .NET Framework 4.0.

3. Dynamic memory allocation

One of the most important resources of a computer it's its memory, reason why efficient algorithms are required for its use and management. Memory allocation can be made both static by the compiler and dynamic during program execution (runtime).

There are four areas of memory used by an application:

- code area (text) – the machine language instructions of the program (program code);
- data area – global variables defined in a program;
- stack – temporary data (local variables) of a program;
- heap-ul – dynamic memory.

Heap is the memory area dedicated to the dynamic allocation of memory. When at compile time is unknown how much memory is required to be allocated for a variable or data structure, the memory is dynamically allocated in the heap, during program execution. Heap is an area that changes size and, which is not managed by the compiler. Is the programmer's duty to know how much memory must allocate and when he must free it.

Common problems in most of programs are related to the loss of references at allocated areas (memory leaks) or reference at unallocated areas or insufficient allocated (invalid access).

4. Garbage collector

In programming languages such as C#, Java, etc the programmer has not anymore the responsibility of distroying its objects, because whereas running a program, also runs a process that deals with the destruction of objects that are no longer used. This process is called garbage collector.

An object is removed from memory by the collection process when there is no longer a reference to it. There are two ways of destroying the references:

[25] http://msdn.microsoft.com/en-us/library/ff650691.aspx

- natural , when the variable goes out of his field of visibility, for example at the end of the method in which it was declared;
- explicit, if we assign null value to the variable concerned.

How does garbage collector work? Garbage collector is a low priority process that runs periodically, dynamically scans the occupied memory by the running program and marks those objects that have direct or indirect references. Once all objects have been completed, those remained unmarked are automatically removed from memory[26]. The functioning mode of the garbage collector can be seen in figure 1.

Must be mentioned the fact that garbage collector supports three generation of objects, as we can see in figure 2. The higher the number of a generation is, the older are the objects of that generation. Also, the objects from the same generation have approximately the same age.

5. String vs. StringBuilder

The code sequence made and presented in this chapter is described in terms of the manner of allocating the heap memory, but also garbage collector's behaviour. To highlight these aspects, we used CLR Profiler tool, presenting the specific differences between String and StringBuilder classes.

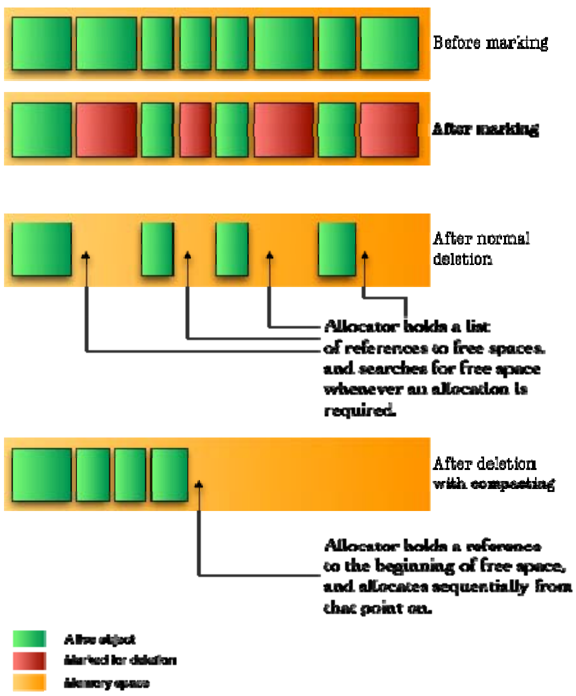
Whether they use directly the *String* class or its alias, *string*, some programmers doesn't know what happens exactly when they concatenate strings. Thus, we will compare the memory allocated for concatenation, using by turn the String class, then the StringBuilder class. In figure 3 and figure 4 respectively, are shown the two methods used in the profiled application.

To profile an application, you must open CLR Profiler, press the Start Application button from the interface shown in figure 5, then choose the executable file of the application to profile. Before starting the application, check the options Allocations and Calls to view both allocations and calls made while running the application.

Figure 6 and figure 7 respectively, present the data resulting from profiling the application. We can notice that the memory allocated using the String class is much bigger (3,119,910 bytes) in comparison with the StringBuilder class (305,670 bytes). This is due to the fact that every time we concatenate, in case of the String class, which is said to be immutable, a new object is created, fact that seriously affects the application's performance.

Also, for a more detailed view, we can see the Histogram Allocated Types (figure 8 and figure 9 respectively) which allows us to view the types of objects that were allocated during program execution.

Figure 1. The functioning mode of the garbage collector



[26] Frăsinaru, C., "Curs practic de Java", Iași , Editura Universității "Alexandru Ioan Cuza";

Figure 2. The garbage collector's generations

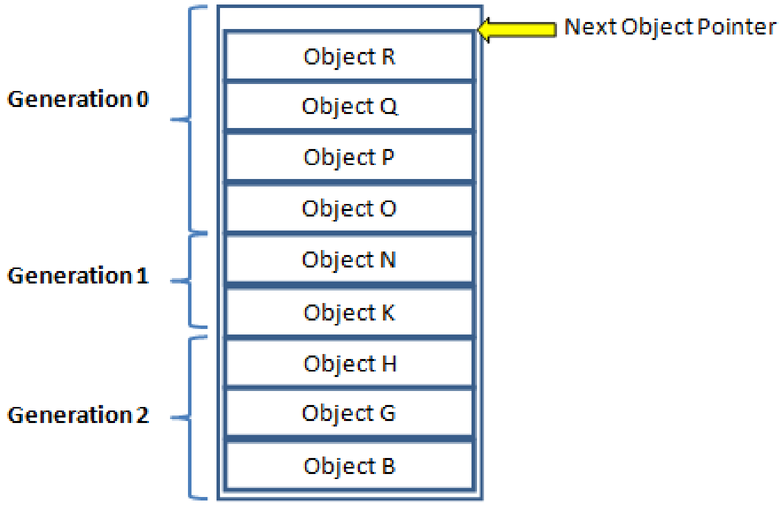


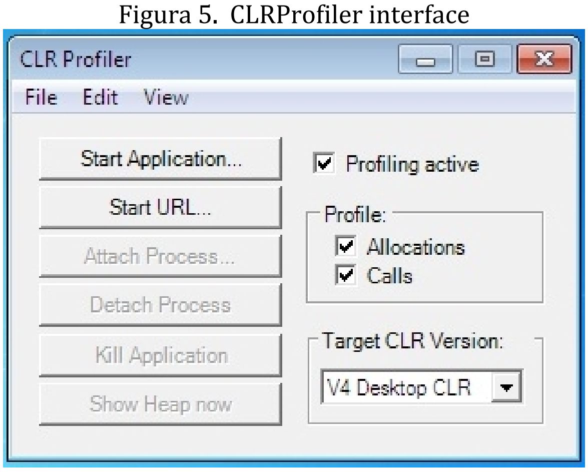
Figure 3. ConcatenareString method

```
private static string ConcatenareString()
{
    string sirCaractere = null;
    for (int i = 0; i < 1000; i++)
    {
        sirCaractere = sirCaractere + i;
    }
    return sirCaractere;
}
```

Figura 4. ConcatenareStringBuilder method

```
private static string ConcatenareStringBuilder()
{
    StringBuilder sirCaractere = new StringBuilder();
    for (int i = 0; i < 1000; i++)
    {
        sirCaractere = sirCaractere.Append(i);
    }
    return sirCaractere.ToString();
}
```

Figura 5. CLRProfiler interface



In case of the histogram corresponding to the String class, we can see that from approximately 3 gigabytes allocated, more than 2 gigabytes are allocated for String objects, while in case of the code which uses the StringBuilder class, the allocated memory is smaller, approximately 300,000 bytes, of which only 42,488 for String objects.

To avoid memory overflow is recommended to use the StringBuilder class, that will create an object only once, then using the Append method will be added strings to the respective object.

Also, in figures 6 and figure 7, we can view the size of the garbage collector's generations. It must be known that all the objects created at runtime, firstly can be found in Generation 0, then the oldest one pass to Generation 1. If one generation is full, the oldest objects are moved to Generation 2.

In case of using the StringBuilder class, we can notice that the size of generations is unknown, meaning that all objects are allocated in Generation 0 and freed also in this generation, while in case of the String class, many objects are moved in Generations 1 and 2, which is not normal.

Then why we should use the String class when we can use the StringBuilder class? Because when we initialize a StringBuilder the application's performance is affected. Also, many facilities provided by String, cannot be used with StringBuilder. Therefore, using the StringBuilder class is indicated only with concatenation, otherwise we can use the String class.[27]

Figura 6. The resulting data from profiling the application that uses String

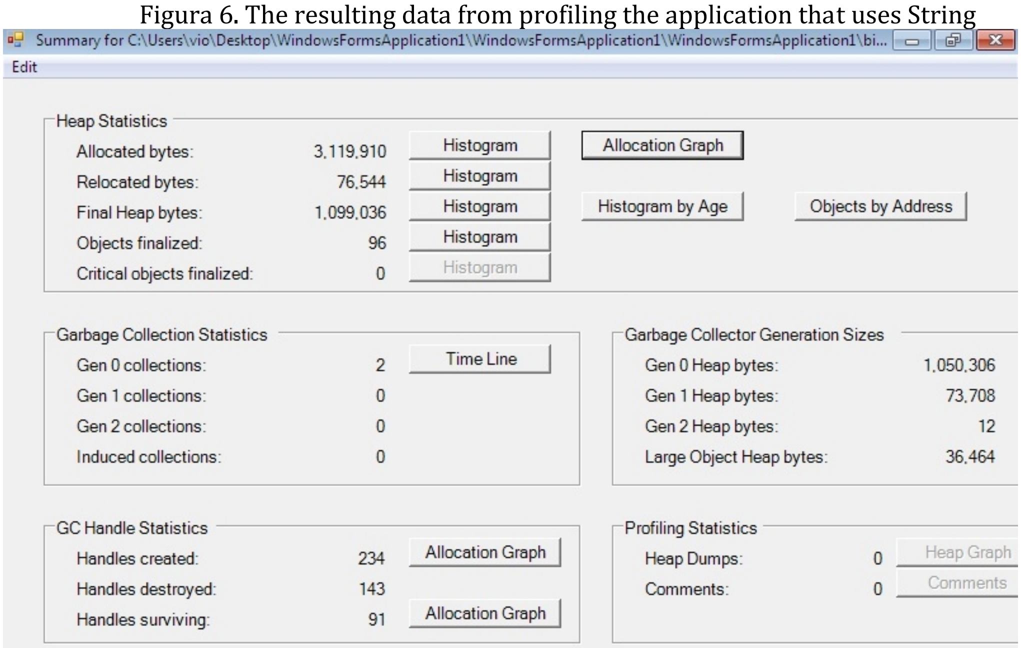
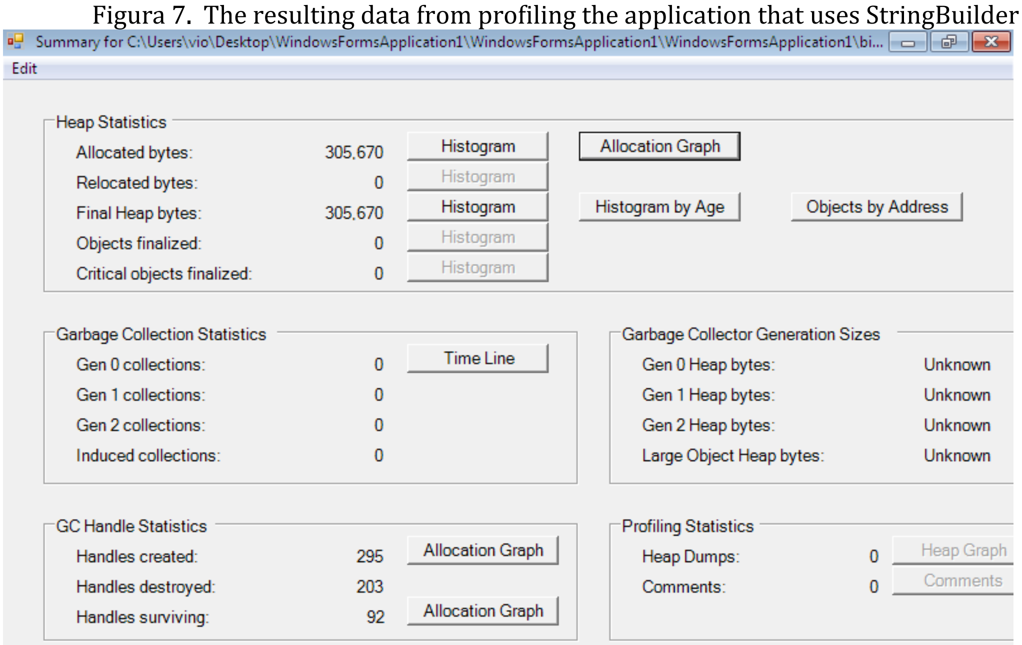


Figura 7. The resulting data from profiling the application that uses StringBuilder



[27] http://www.codeproject.com/Articles/6771/String-Vs-StringBuilder-C

Figura 8. Histogram by size for allocated objects – using String

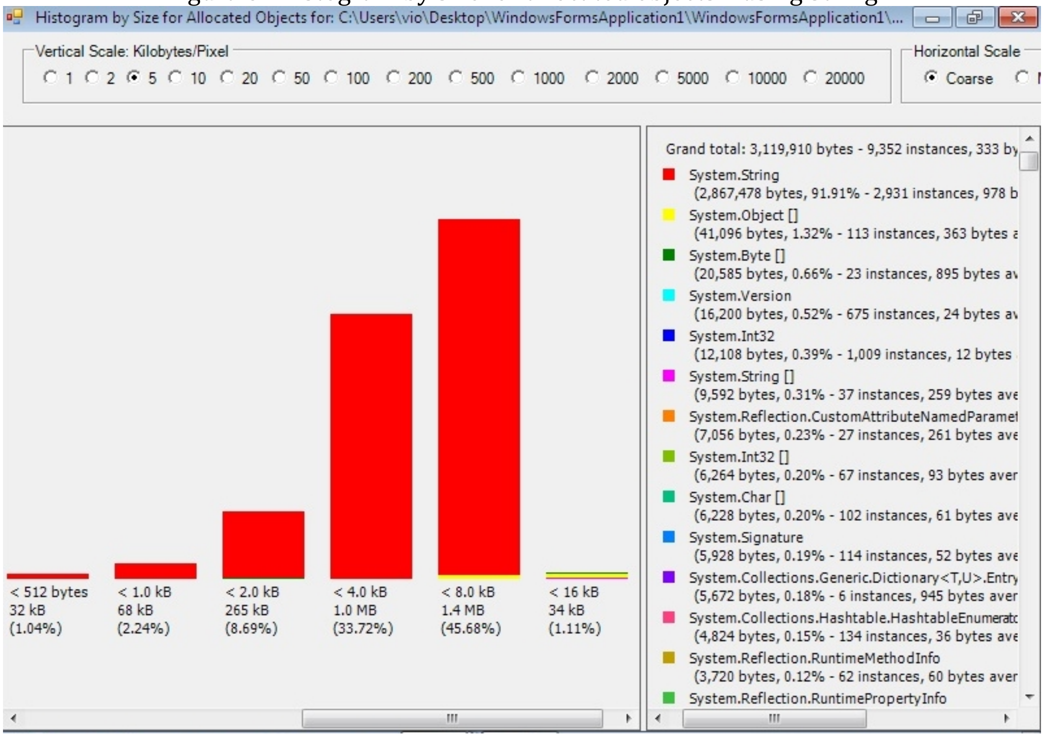
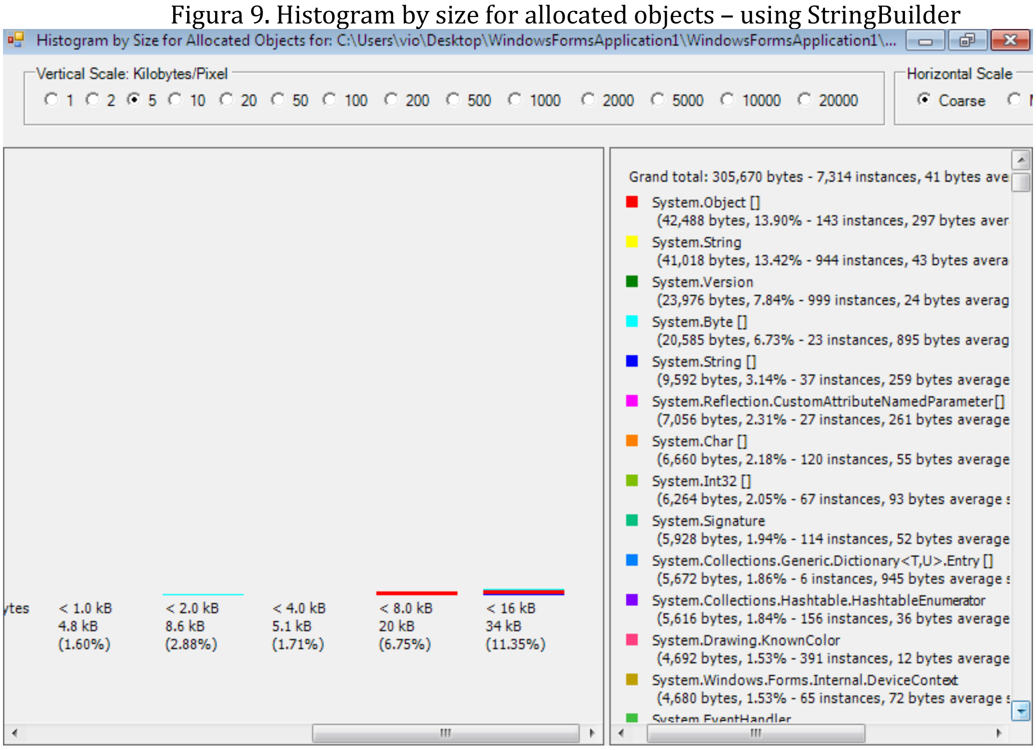


Figura 9. Histogram by size for allocated objects – using StringBuilder



6. Conclusions

The specific activities of each organization are surrounded by an impressive amount of information. For this reason, we need information systems capable of managing and processing them efficiently and providing the final user, relevant information, at the appropriate time. Thus, it becomes very clear that these information systems must be designed and implemented so as to efficiently use each physical resource of the computer.

In this context, during this paper has been presented a simple application, which was analyzed using CLR Profiler tool, highlighting the importance of a proper management for the allocated memory of an application, but also the importance of a profiler able to indicate the potential areas of code that may cause problems (inefficient allocation of memory). As shown in this paper, CLR Profiler can provide a number of advantages and benefits among which the most important refer to the identification of problems related to the loss of references to allocated areas or unallocated areas or insufficient allocated.

References

1. *Frăsinaru ,C., "Curs practic de Java", Iași , Editura Universitătii "Alexandru Ioan Cuza ".*
2. *Griffith Rean, Kaiser Gail, Manipulating managed execution runtimes to support self-healing systems, Proceedings of the workshop on Design and evolution of autonomic application software, ISBN:1-59593-039-6, pp. 1-7.*
3. *Kansal Aman, Zhao Feng, Fine-grained energy profiling for power-aware application design, ACM SIGMETRICS Performance Evaluation Review, Volume 36 Issue 2, September 200m pp. 26-31.*
4. *http://dl.acm.org/citation.cfm?id=1190224*
5. *http://elf.cs.pub.ro*
6. *http://msdn.microsoft.com/en-us/library/ee851764(v=vs.110).aspx#performance_analysis_tools*
7. *http://stackoverflow.com/*
8. *http://vsh.ro*
9. *http://www.codeproject.com*