2018

Performance of Memory Deallocation in C++, C# and Java

Luís Henriques
Polytechnic of Coimbra –ISEC, a21260884@alunos.isec.pt

Jorge Bernardino
Polytechnic of Coimbra –ISEC, jorge@isec.pt

Desempenho de Desalocação de Memória em C++, C# e Java

Performance of Memory Deallocation in C++, C# and Java

Luís Henriques, Polytechnic of Coimbra –ISEC, Dept. of Computer and Systems Engineering, Coimbra, Portugal, a21260884@alunos.isec.pt

Jorge Bernardino, Polytechnic of Coimbra –ISEC, Dept. of Computer and Systems Engineering, Coimbra, Portugal, jorge@isec.pt

Resumo

Gestão de memória é uma componente essencial de qualquer aplicação. Existem estratégias para desalocar memória, como ponteiros inteligentes, garbage collectors e contagem de referências. Para minimizar a sobrecarga associada à sua execução, as linguagens de programação implementam estratégias de otimização nos seus procedimentos de desalocação de memória. O objetivo deste artigo é comparar o desempenho de três dessas estratégias: os sistemas de gestão inteligente de ponteiros de C++ e os garbage collectors de Java e C#. Para medir o seu desempenho, criámos duas aplicações destinadas a testar alocações de memória profundas e superficiais. Essas aplicações alocam um total de 100.000 objetos. Reproduzimos as aplicações em C++ usando as classes unique_ptr e shared_ptr e em C# e Java, para um total de oito aplicações. O sistema de garbage collection de C# superou consistentemente os outros devido às suas otimizações de desalocação assíncrona da memória.

Palavras-chave: Desempenho; memória; desalocação profunda e superficial

Abstract

Memory management is an essential component of any application. There are strategies used to deallocate memory such as smart pointers, garbage collectors and reference counting. To minimize the overhead associated with their execution, higher level coding languages tend to implement optimization strategies on their memory deallocation procedures. The goal of this paper is to compare the performance of three memory deallocation strategies: C++'s smart pointer management systems and C# and Java's garbage collectors. To measure their performance, we created two simple applications aimed at testing deep and shallow memory allocations. These applications allocate a total of 100.000 objects on the heap. We reproduced these applications in C++ using both unique_ptr and shared_ptr classes, as well as in C# and Java, for a total of eight applications. C#s garbage collection system consistently outperformed the others due to its optimized procedures of asynchronously deallocating memory.

Keywords: *Performance; memory; deep and shallow deallocation*

1. INTRODUCTION

Performance tests and comparisons between coding languages are a common topic in computer science research. That is due to the fact that the choice of coding language has an enormous impact on the development and efficiency of a given planned application.

Currently, there is a debate about the advantages of managed languages, like Java and C#, versus the high performance of traditional coding languages such as C++ (Gherardi, 2012; Nikishkov, 2003; Sestoft, 2010). The argument is that managed languages are not only safer and easy to use, but they achieve good performance due to the optimizations built in their Virtual Machines (VM) (Sestoft, 2010). Furthermore, managed languages have an enormous compatibility advantage: compiled languages, such as C++, are translated into machine code through a compiler, which generates files that can be executed directly by the CPU. This means that these applications are platform depended, and must be compiled for every single computing platform; managed languages, on the other hand, are interpreted by a Virtual Machine. Even though the VM itself is built on a compiled language, and is, consequently, platform dependent, it is able to interpret any application coded with the appropriate language. Hence, any Java program can be interpreted by any Java Virtual Machine (JVM), and any C# application is able to be interpreted by the Common Language Runtime (CLR), without the need to recompile the application for every single computing platform.

There are many studies that aim to compare the performance of both language types (Gherardi, 2012; Sestoft, 2010; Singer, 2003; Vivanco, 2002). Yet most are centred on small tasks like expression evaluation and memory management, which are extremely important. However, the larger tasks such as memory deallocation and garbage collection, which are widely employed and very frequently performed by the computer, are not compared as often. We consider that it is important to address these fundamental ordinary tasks, since they are a constant presence in any application.

There is a wide range of strategies used to manage and deallocate memory such as smart pointers, garbage collectors and reference counting, just to name a few. There are also new strategies being developed. We take the example of the compact-fit memory management system, which aims to improve performance by keeping memory fragmentation predictable (Craciunas et al., 2008). The main goals of such strategies are to free up memory space and prevent memory leaks. There are also strategies that aim to allocate memory in an optimal fashion, so it can be accessed efficiently, and thus, reduce the time it takes to manage it (Bertels, 2009).

In low level languages, the programmer has the responsibility to explicitly deallocate memory and prevent memory leaks. C++11, for example, tries to help the developers by adding a few classes that encapsulate some memory management behaviour. That is the case with the smart pointer classes.

The *unique_ptr* class guarantees that there is only one pointer referencing a location in memory. When a unique pointer is deleted, reallocated, or leaves scope, it simply deallocates the memory space it was pointing to. The *shared_ptr* class, on the other hand, uses reference counting to keep a record of all pointers that are pointing to a location in memory. When the last pointer is deleted, it deallocates the respective memory space. Such strategy can lead to memory leaks by leaving a reference loop on the heap, so it is up to the programmer to prevent them. These are two different memory management strategies, with different overhead costs.

As for high-level coding languages, these tend to implement some sort of automatic management system, such as a garbage collector: a memory management procedure that is responsible for automatically deallocating memory once it identifies which variables need to be kept or wiped from memory. These approaches implement measures that prevent reference loops and other memory leaks, which have an associated overhead as well, making them not ideal, but they are still the most widely used solutions for automatic memory management. To minimize the performance overhead, higher level coding languages tend to implement optimization strategies in order to facilitate the memory deallocation procedures.

It is important to note that there are always performance tradeoffs for each strategy. Still, we do not know which strategy works best in regards to performance, nor if the garbage collecting procedures' overhead is sufficiently minimized by its optimizations.

This work aims to understand which coding language deallocates memory the fastest. Our goal is to measure the impact that the choice of coding language has on the performance of the memory deallocation system of a given application.

The remainder of this paper is organized as follows. In section 2, we present an overview of the most relevant research on performance of coding languages. Section 3 describes the methodology used for the assessment. In section 4 we present the experimental evaluation and in section 5 we discuss the results. Finally, in section 6, we present some conclusions while providing some insight for future work.

2. RELATED WORK

The performance comparison of coding languages is extremely relevant to computer science and engineering, since it aims to improve overall throughput, eliminate performance bottlenecks and, most importantly, anticipate poor performance situations (Wescott, 2013).

In robotics, there is the assumption that compiled languages, mainly C++ and C, have a better performance than interpreted languages. Due to the fact that low level languages are less compatible, more laborious, and harder to implement, it would be preferable to use a higher-level language such as Java or C#. With that in mind, Gherardi et al. (2012) challenged the status quo by considering the performance of Java versus C++. The goal was to understand if Java is, in fact, a viable alternative in robotics. To achieve it, the authors created similar applications for both Java and C++ and tested for memory allocation and retrieval performance through the management of large collections of data (three point vectors, which refer to a point in a three dimensional space). They also studied the performance when evaluating logical propositions in both languages. They concluded that Java performance has evolved considerably and is better than what is reported in the literature. They also concluded that using a server compiler for a long running application in Java greatly improves the application's performance, making it a viable alternative to C++. Adding the advantages of portability and maintainability so characteristic of managed languages, they make a good argument for using Java instead of a low-level compiled coding language.

As for comparing the performance of interpreted languages themselves, a study by Singer (2003) that aimed to compare the execution time performance of JVM 1.4 applications in Java versus CLR 7.0 (.NET Common Language Runtime) applications in C#, concluded that there is virtually no difference between both runtime environments.

In another study that aimed to compare the performance of Java and C++ in the analysis of image-based biomedical data (Vivanco, 2002), an undertaking which usually requires huge datasets of functional magnetic resonance imaging (fMRI) and magnetic resonance spectroscopy (MRS) images, the authors implemented the same data model and computational algorithms in C++ and Java and assessed their performance. Even though they concluded that C++ still outperforms Java, they also determined that there are optimizations present in the Java Virtual Machine that are quickly closing the performance gap between managed and compiled coding languages. These optimizations are, obviously, extremely important when studying performance.

Such optimizations are presented in more detail in a report written by Sestoft (2010), in which the author aims to compare numeric performance in Java, C# and C, and argues that Java and C# compete well with C++ and C on numeric code due to the embedded optimizations of the Java Virtual Machine. For computations involving arrays or matrixes of floating-point numbers, the situation is not so favourable. This is due to the fact that compilers for C and C++ make a serious effort to optimize loops for array access, which is not so prevalent on the Just in Time (JIT) compilers of the Java and C# runtime systems. Furthermore, the authors explain that "in C# and Java there must be an index check on every array access, and this not only requires execution of extra instructions, but can also lead to branch mispredictions and pipeline stalls on the hardware, further slowing down the computation" (Sestoft, 2010). Their conclusion is that these compiler optimizations can make a serious difference in performance and, for that very reason, the choice of execution environment, whether it is the JIT compiler or the chosen VM, is of extreme importance. This conclusion is also confirmed by Nikishkov et al. (2003), which suggest that the Java Virtual Machine 1.2 (JVM 1.2) produces the best performance of all JVMs.

While it is true that compiler optimization techniques improve the performance of applications, their evolution revealed a second tier of performance bottlenecks: the standard libraries themselves.

That is the case presented in a paper written by Zhang J., Lee J. and McKinley P. K. (2005), in which the authors explain how they improved the performance of the studied applications "by over a factor of 4 on average, and by a factor of 27 in the best case". They did so by improving the standard Java Piped I/O Stream library. They found out that "the Java pipe library is implemented in a very inefficient way" and, due to the fact that they are frequently used, "their efficiency may significantly impact on the performance of many programs". Developing these library optimizations may require effort, but they could have a superior impact on performance when compared to conventional compiler or code optimizations. Each coding language has their own set of standard libraries, each with its own unique optimizations. They are conceptually similar to traditional optimizations and may include: the option for specialized routines, which would eliminate unnecessary library calls and eliminate redundant computations; computations at compile-time; the exploitation of special cases; code scheduling; and transformations. All these ideas are presented in full detail in a paper by Guyer, S. Z. and Lin, C. (2000), in which the authors explore the implications of library-level optimizations. It is also possible to integrate memory management extensions to allow finer control over a memory management system. A paper by Beebee (2001) reports the experience of implementing such extensions in the Real-Time Specification for Java. These extensions give the programmer the ability to control the memory management behaviour of a given Java application, and predict how memory is both allocated and deallocated.

Garbage Collection algorithms are themselves a target for optimizations. With the improvements in the speed of microprocessors, it is becoming very common for developers to try to achieve higher performance in their application through multithreading. This allows parallel execution of tasks by distributing the workload through multiple threads. Many of these garbage collection algorithms use this multithread workload distribution as well to manage memory deallocation. The JVM uses a protocol, Completely Fair Scheduler, to evenly distribute the CPU time to all running processes in the queue. However, synchronizations between threads generate a huge amount of overhead, which creates a challenge in concurrent copying garbage collection to "design a concurrent copying protocol with a small but sufficient number of synchronizations" (Ugawa, 2017). This is perceivable in other studies, such as the one by Qian J. et al. (2016) in which the authors reviewed parallel garbage collection algorithms and concluded that "the garbage collector performs worse if the application’s workload is evenly distributed among threads" than if it was using a pseudo first-in-first-out protocol to schedule them. In large multicore systems, this greatly improves scalability of the garbage collection process, while reducing the overall time it takes to be performed.

One common pattern employed in memory deallocation systems is the generational garbage collector. This design allows for the garbage collection system to distribute the objects in generations, or "time intervals", and make decisions using their lifecycle as criteria. By dividing the heap space in different sections the system avoids having to manage the heap space in bulk, which would be very inefficient. Usually, the heap space allocated for the most commonly used section, or generation 0, is fairly small when compared to the other sections. The system then allocates new objects in the first heap section, marking them as generation 0 objects. Once this generation 0's heap space is full, the system sweeps the section and tries to free memory by looking for unused objects. Any other objects that are still being used and survived the first sweep are then reallocated to the next section. Once this second section is full, it is swept as well, along with all previous sections, and its objects are reallocated to the next section. This reallocation process is possible because, effectively, every object exists as a pointer, which then points to a "table of pointers" that contains the real pointers. This allows for any object to be moved between heap sections by updating only one of these real pointers.

Since, as stated by the weak generational hypothesis, "most objects die young" (Qian, 2016), the collection of objects from the youngest generation occurs much more frequently than the collection of objects from the older generations. This allows for the improvement of the garbage collector's memory management process and improves performance by preventing frequent unnecessary exhaustive heap analysis. Most times, only the small, generation 0 heap space needs to be analyzed and swept.

As for comparing garbage collection behaviour, another paper published by Lengauer, P. et al. (2017) that compares two different garbage collection algorithms, GC G1 and ParallelOld GC, concludes that there is a huge difference regarding the collection management process of both algorithms. The authors state that, due to the fact that GC G1 can select which heap regions to collect, it can perform less collection cycles, since it selects regions with more garbage, which results in more memory being freed each time. GC G1 can also include regions of old generations in its minor collection cycles, which reduce the effect of old generation dead objects keeping young objects alive, and allows for this algorithm to only perform major collection cycles as emergency actions. In contrast ParallelOld GC can only collect older generation objects in major collection cycles, which hinders performance and heap optimization. The authors also point out that GC G1 does not perform well when managing small heap sizes, which leads to the conclusion that many optimizations imply some sort of tradeoff, and many of them are more or less viable given certain conditions.

There is also a point to be made regarding code optimization techniques. Even though that is dependent on the skill level of the programmers themselves, it is shown that the use of a small amount of unsafe code in C# (Sestoft, 2010), or the use of code tuning techniques in Java (Nikishkov, 2003), could significantly improve the performance of applications in both languages to the point in which they can compete with C and C++. An important idea to keep in mind when coding for performance.

Our work aims to understand the impact of any possible optimizations in the memory deallocation systems of C++, C# and Java, how they handle different coding structures, and in what measure they could have an impact on the performance of each one.

3. METHODOLOGY

In order to do the performance evaluation test, we decided to take an "unadorned" approach, since simplicity in performance testing is the key. It reduces the number of interfering variables and helps us guarantee the reliability of the results. As such, we closed all running applications, leaving only the ones required by the operative system's basic procedures. This way, we will hopefully prevent any interference from other applications, as there will be a limited number of operations competing for the CPU's attention. We also kept the code to a bare minimum, so we could safely replicate the application in different coding languages without running the risk of writing code that is significantly different for each one. Our goal was to use the exact same code as much as possible by using equivalent operations from each coding language.

The experimental setup for our tests was as follows:

- Computer: HP Pavilion dv6 Notebook PC;
- Processor: Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz, 2001 MHz, 4 Cores, 8 Logical Processors;
- RAM: 8GB;
- Operating System: Windows 7 Home Premium, 64 Bit.

We divided the experiments in two phases in order to test two distinct approaches to memory deallocation: 1) deep deallocation and 2) shallow deallocation.

The first phase, the one testing deep deallocation, aims to assess the performance of each language when deallocating a large chain of objects. With this test we want to understand in what measure each coding language can identify objects that are dependent on other objects, and evaluate how long each language's memory deallocation procedures take to recognize that they can release a full chain of dependent objects from memory. We intend to perceive the consequences of any optimizations present on each language's standard memory deallocation procedures. Are the memory deallocation algorithms able to understand these object dependencies in an efficient way? How is the performance of each coding language's memory deallocation systems regarding the management of these dependencies? Do they release the full chain of objects efficiently?

The second phase of our experience aims to test shallow deallocation. As such, all objects coexist on the same level and are deallocated in parallel. Contrasting with our first test phase, we mean to compare the performance of each coding language when releasing a large set of independent objects from memory. This should represent the standard set of operations performed by any memory deallocation strategy and, as such, it is a good target for any possible optimizations. We mean to perceive the consequences of such optimizations by running the applications against each other. In other words, this should equate to a simple comparison of performance between the standard memory deallocation strategies of each of the coding languages we are going to test.

In each of our test phases, we assessed the performance of four deallocation strategies: i) C++'s unique_ptr class; ii) C++’s shared_ptr class; iii) C#’s standard garbage collector; and iv) Java's standard garbage collector.

For C++, we decided to allocate our objects on the heap instead of the stack, since both C# and Java always allocate memory on the heap as well. Our goal was to replicate the inner workings of all three languages as much as possible.

We then coded a total of eight simple applications, four for each of the two phases. These applications create 10 sets of 10.000 objects, for a total of 100.000 objects, and then deallocate them from memory. We isolated the deallocation process for each coding language and measured how long this specific process took to be performed. Each time we ran a cycle, we deallocated the memory beforehand, so that we would guarantee that any remaining allocation would not interfere with the measurements. The exception is, of course, C++, since we target what we want to deallocate.

To gather the data and assess the test results, we avoided using the performance profiler from each application's IDE. These profilers would have to inject some procedures for the profiling process to take place, which could influence our results. The overhead of such calls could be costly compared to the application's actual time. As such, we simply ran some timers, with as much level of precision as possible, that were started immediately before, and stopped immediately after, each deallocation process. The duration of each process was then simply printed to the console.

We ran each application three times and took note of the maximum, minimum and average results for each cycle.

For our first test, deep deallocation, we created the TestModel class in our first four applications. This class has a recursive constructor through which its objects can replicate themselves. The goal of each object is simply to exist on the heap and replicate itself once. The replicated object then replicates itself as well, generating a chain of objects with a predefined limit (see Figure 1). This way, we are able to test how each memory deallocation strategy reacts to a chain of objects. For C++ we simply delete the first parent object at the chain's beginning. As for Java and C#, we leave the process criteria to their garbage collectors. This will allow us to perceive any potential optimizations in each language's deallocation strategy.

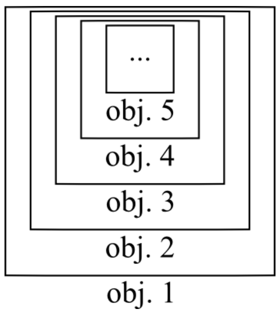

Figure 1 – Deep deallocation.

For our C++ test, we used Visual Studio 2017 Community Edition 15.5.2 with C++11. To deallocate our objects from memory, we simply used the .reset() function on the main parent object, which starts the memory deallocation process for all children objects as well.

The code we considered more pertinent to reproduce our experiment is as follows:

TestModel's header for the C++ unique_ptr application:

```
class TestModel
{
private:
    unique_ptr<TestModel> myPtr;

public:
    TestModel();
    TestModel(int count);
    ~TestModel();
};
```

TestModel's body for the C++ unique_ptr application:

```
TestModel::TestModel(int count)
{
    if (count > 0) {
        count--;
        myPtr = unique_ptr<TestModel>(new TestModel(count));
    }
}
```

Main code for the C++ unique_ptr application:

```
int main()
{
    unique_ptr<TestModel> pTestModel;

    cout << "Press enter to create objs" << endl;
    cin.get();

    clock_t start;
    double runDuration;
    double totalDuration = 0;

    for (int i = 0; i < 10; i++)
    {
        cout << "beginning cycle " << i << endl;

        pTestModel = unique_ptr<TestModel>(new TestModel(10000));

        runDuration = 0;
        start = clock();
        pTestModel.reset();
        runDuration = (clock() - start) / (double)CLOCKS_PER_SEC;
        cout << "Run time " << runDuration << " secs" << endl;
        totalDuration += runDuration;
    }

    double avgTime = totalDuration / 10;
    cout << endl;
    cout << "- Done -" << endl << endl;
    cout << "Total time " << totalDuration << endl << endl;
    cout << "Average time " << avgTime << endl << endl;
    cin.get();
}
```

Since we considered that C++'s *unique_ptr* class uses a different strategy than *shared_ptr*, and the *shared_ptr* class's reference counting strategy emulates more closely what is actually performed by Java and C#, we developed a version to test the *shared_ptr* class as well. Its code is exactly the same as the *unique_ptr* test, simply replacing *unique_ptr* for the *shared_ptr* class.

As for the C# application, we also used Visual Studio. We targeted .Net Framework 4.6.1 using C# 7.2. The code is as follows:

TestModel for the C# application:

```
class TestModel
{
    TestModel myTestModel;

    public TestModel(int count)
    {
        count--;
        if (count > 0)
        {
        myTestModel = new TestModel(count);
        }
    }
}
```

Main code for the C# application:

```
static void Main(string[] args)
{
    Stopwatch gcTimer = new Stopwatch();
    Console.WriteLine("Current memory: " + GC.GetTotalMemory(false));
    Console.WriteLine("Press any key to create objs");
    Console.ReadLine();

    TimeSpan totalTime = TimeSpan.Zero;
    long totalTicks = 0;

    for (int i = 0; i < 10; i++)
    {
        GC.Collect();
        Console.WriteLine("beginning cycle " + i);
        Console.WriteLine("Current memory: " + GC.GetTotalMemory(false));

        TestModel tm = new TestModel(10000);
        tm = null;

        gcTimer.Restart();
        GC.Collect(2, GCCollectionMode.Forced, true);
        gcTimer.Stop();

        Console.WriteLine("Current memory: " + GC.GetTotalMemory(false));

        totalTime += gcTimer.Elapsed;
        Console.WriteLine("Garbage collection took {0} secs", gcTimer.Elapsed);

        totalTicks += gcTimer.ElapsedTicks;
        Console.WriteLine("Garbage collection took {0} ticks",
        gcTimer.ElapsedTicks);
        Console.WriteLine("--");
    }

    Console.WriteLine("GC took " + totalTime + " seconds");
    Console.WriteLine("GC took " + (totalTime.Seconds / 10) + " seconds");
    Console.WriteLine("GC took " + totalTicks + " ticks");
    Console.ReadLine();
}
```

Last but not least, we developed the Java application. We used NetBeans 8.2 with Java Development Kit 8u152 x64. The TestModel class follows the exact same logic as the one in C# and, as such, it is not necessary to show the code.

As for the rest of the application, the code is:

Main code for the Java application:

```
public static void main(String[] args) {
    try {
        System.out.println("Press enter to create objs");
        System.in.read();
        long totalTime = 0;

        for (int i = 0; i < 10; i++)
        {
        System.gc();
        System.out.println("beginning cycle " + i);

        TestModel t = new TestModel(10000);
        t = null;
        long beginTime = System.nanoTime()
        System.gc();
        long endTime = System.nanoTime()

        long loopTime = endTime - beginTime;
        totalTime += loopTime;
        System.out.println("Garbage collection took " + loopTime);
        System.out.println("--");
        }

        System.out.println("GC took " + totalTime + " milliseconds");
        System.out.println("Average time " + (totalTime / 10) + " milliseconds");
        System.in.read();

    } catch (IOException ex) {
        System.out.println(ex.getMessage() + " " + ex.getStackTrace());
    }
}
```

For our second test, 2) shallow deallocation, we used a similar process. We created a simple TestModel class for our four shallow deallocation applications as well. We then adapted the code in order to make it create and keep in memory 10 sets of 10000 objects at a time in parallel, which would be deleted immediately after (see Figure 2). We also measured how long each deallocation process took.

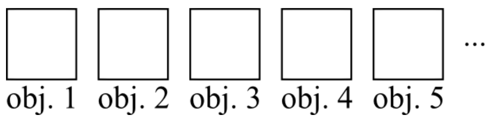

Figure 2 – Shallow deallocation.

For C++, we tested which process would be the fastest: placing the whole loop inside the stopwatch cycle, or starting and stopping the stopwatch each time we run a cycle (which is 10 x 10.000 times). Turns out that placing the whole loop inside the stopwatch cycle is faster, but the difference is not relevant. For that reason, and to maintain simplicity, we decided to start our stopwatch, run a whole cycle of 10.000 deallocations, and only then stop the stopwatch.

Just like with the first phase of tests, we used the same IDEs and measured our timings in the same way.

Main code for the C++ unique_ptr application:

```
int main()
{
    cout << "Press enter to create objs" << endl;
    cin.get();

    clock_t start;
    double runDuration;
    double totalDuration = 0;

    for (int i = 0; i < 10; i++)
    {
        unique_ptr<TestModel> pTestModel[10000];
        cout << "beginning cycle " << i << endl;
        for (int j = 0; j < 10000; j++)
        {
            pTestModel[j] = unique_ptr<TestModel>(new TestModel());
        }

        runDuration = 0;
        start = clock();

        for (int k = 0; k < 10000; k++) {
            pTestModel[k].reset();
        }

        runDuration = (clock() - start) / (double)CLOCKS_PER_SEC;
        cout << "Run time " << runDuration << " secs" << endl;
        totalDuration += runDuration;
    }

    double avgTime = totalDuration / 10;
    cout << endl;
    cout << "- Done -" << endl << endl;
    cout << "Total time " << totalDuration << endl << endl;
    cout << "Average time " << avgTime << endl << endl;
    cin.get();
}
```

Once again, the code is the exact same for our *shared_ptr* class test, simply replacing the *unique_ptr* instances with *shared_ptr* ones.

Main code for the C# application:

```
static void Main(string[] args)
{
    Stopwatch gcTimer = new Stopwatch();
    Console.WriteLine("Press any key to create objs");
    Console.ReadLine();

    TimeSpan totalTime = TimeSpan.Zero;

    for (int i = 0; i < 10; i++)
    {
        GC.Collect();
        Console.WriteLine("beginning cycle " + i);
        TestModel[] modelList = new TestModel[10000];

        gcTimer.Restart();
        GC.Collect(2, GCCollectionMode.Forced, true);
        gcTimer.Stop();

        Console.WriteLine("GC took {0} secs", gcTimer.Elapsed);
        Console.WriteLine("--");
        totalTime += gcTimer.Elapsed;
    }
    Console.WriteLine("GC took " + totalTime + " seconds");
    Console.ReadLine();
}
```

The Java application follows the same sequence, and its code is:

Main code for the Java application:

```
public static void main(String[] args) {
    try {
        System.out.println("Press enter to create objs");
        System.in.read();

        long totalTime = 0;
        for (int i = 0; i < 10; i++)
        {
            System.gc();
            TestModel[] modelList = new TestModel[10000];
            System.out.println("beginning cycle " + i);
            modelList = null;

            long beginTime = System.nanoTime();
            System.gc();
            long endTime = System.nanoTime();

            long loopTime = endTime - beginTime;
            totalTime += loopTime;
            System.out.println("Garbage collection took " + loopTime + "ms");
            System.out.println("--");
        }

        System.out.println("GC took " + totalTime + " ms");
        System.out.println("Average time " + (totalTime / 10) + " ms");
        System.in.read();

    } catch (IOException ex) {
        System.out.println(ex.getMessage() + " " + ex.getStackTrace());
    }
}
```

4. Performance Evaluation

In order to evaluate the performance of each memory management system, we ran each application three times. For each run we considered the time it took to perform the deallocation process for each of the 10 sets of 10.000 objects and took note of the fastest run (minimum time). We consider minimum time the most important metric, because any excess delay in each run can be attributed to the operating system's background processes. We also took note of the full processes' duration for the 100.000 objects which, when divided by 10, represents the average time for deallocating each 10.000 objects. All measures are presented in milliseconds.

For our first test, 1) deep deallocation, the results are presented in Table 1.

As for our second test, 2) shallow deallocation, the results are presented in Table 2.

	MIN			MAX			AVG		
RUN	*1st*	*2nd*	*3rd*	*1st*	*2nd*	*3rd*	*1st*	*2nd*	*3rd*
UNIQUE_PTR	0.6	0.6	0.6	0.76	0.86	0.92	0.67	0.73	0.76
SHARED_PTR	0.9	0.9	1	1.12	1.18	1.18	1.01	1.04	1.09
C#	0.03	0.04	0.04	0.05	0.04	0.04	0.04	0.04	0.04
JAVA	4.9	4.8	4.7	6.9	5.2	5.3	5.9	5	5

Table 1 – Performance of Memory Deallocation Procedures for Deep Deallocation (in milliseconds).

	MIN			MAX			AVG		
RUN	*1st*	*2nd*	*3rd*	*1st*	*2nd*	*3rd*	*1st*	*2nd*	*3rd*
UNIQUE_PTR	0.5	0.5	0.5	0.58	0.62	0.54	0.54	0.56	0.52
SHARED_PTR	1.1	1.2	1.1	1.2	1.22	1.35	1.15	1.21	1.22
C#	0.03	0.03	0.03	0.05	0.04	0.04	0.04	0.04	0.04
JAVA	4.8	4.8	4.6	5.8	5.4	5.6	5.3	5.1	5.1

Table 2 – Performance of Memory Deallocation Procedures for Shallow Deallocation (in milliseconds).

5. Discussion of Results

Our results show that, for deep deallocation, C#'s garbage collection system outperforms C++ *unique_ptr* class by 0.67 milliseconds, on average. It also outperforms C++'s *shared_ptr* class by 1 millisecond and Java by 5.26 milliseconds, which is surprising. We didn't expect C#'s garbage collector to outperform Java's by such extent. Let alone outperform C++'s smart pointers. Being a low-level coding language, C++ is considered to be one of the fastest languages. It is pre-compiled, which means it needs no effort to be interpreted at runtime, and it is properly optimized. The fact that smart pointers even exist is evidence of the developer's care for performance.

As for shallow deallocation, C#'s garbage collector outperforms all others as well, with an average difference of 0.5 milliseconds when compared to C++'s *unique_ptr* class, 1.15 milliseconds when compared to C++'s *shared_ptr* class and a full 5.12 milliseconds when compared to Java's standard garbage collector. After retrieving our results we measured the application's times with the IDE's profilers and the outcomes were consistent with our initial results. With this triangulation, we were sure that our measurements reflect our applications' inner workings.

By dividing the heap space, which allows for the reduction of each deallocation task, and then by diluting these small sweep tasks throughout various threads, C#'s standard garbage collection system was able to reduce the impact of the memory management process during runtime, along with its associated overhead, and thus, achieved much better performance.

Note that, as mentioned by Qian, J. et al. (2016), "the garbage collector performs worse if the application's workload is evenly distributed among threads" because "a large number of concurrent threads create heap allocation competition that can lead to prolonged object lifespans", which opens the door for even further optimizations and improvements on performance,

Our results also show that C++ outperforms Java, which was to be expected, at least for the *unique_ptr* class. This class does not need to search for any additional references, since it assumes a single ownership of the object it points to. C++'s *unique_ptr* class outperforms Java's garbage collector by 4.58 milliseconds in our deep deallocation test, and by 4.63 on the shallow deallocation test. Similarly, elements from the *shared_ptr* class, which need to do reference counting when being reset, outperform Java's standard garbage collector by an average of 4.25 milliseconds on the deep deallocation test, and an average of 3.97 milliseconds on the shallow deallocation test.

We also tested the performance of these strategies with deep and shallow memory deallocation. We were able to verify that there is no considerable difference in performance between deep and shallow memory deallocation, even though shallow deallocation presents faster times. The results were consistent in both tests, which show that the performance results were not just a by-product of the structure used for the assessment.

As for developers, it is true that, unless the performance of the application under development is considered critical, and needs to be guaranteed during the full duration of its execution, it is not important to fine tune and optimize the memory management procedures. In fact, it is not even advisable to do so. The standard behaviour and underlying systems of each coding language already try to manage memory as efficiently as possible. Of course there is some disadvantage in trusting memory management to a black-boxed system that is not directly controlled by the programmers themselves (Vassev, 2006), but fine-tuning these systems takes precious, costly, coding time.

However, for most applications, it is important to understand how each one of these deallocation strategies perform since, as stated before, memory management can take up to one third of the total application's execution time. For simpler applications, the ones in which the developers do not perceive any value in trying to customize the memory management operations, C#'s optimizations and strategies seem to be the most useful and the ones that show the best performance overall.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we assessed the performance of memory deallocation processes for C++, Java and C#. We took this opportunity to explore deep and shallow memory deallocation processes, and concluded that C#'s standard garbage collector outperforms both Java's garbage collector and C++'s *unique_ptr* and *shared_ptr* classes' memory management procedures.

From a developer's perspective, it is important to have some idea about how each "out of the box" memory management system works. Using a black box system such as a Garbage Collector can be easier, but not ideal. After all, programming is not about easiness, but about efficiency and full control over the application's behaviour (Vassev, 2006). On one hand, setting up customized memory management systems takes precious development time, but, on the other, inefficient heap management could take out a lot of the application's performance, so this is something developers should consider when developing new applications.

As future work, we intend to further extend these tests by testing a tree structure, in which every model creates two new ones, instead of a simple chain of objects or a wide number of objects on the same level. It would also be alluring to test larger and more complex objects, which would occupy the full first heap section defined by a generational garbage collector. We are also interested in running the tests with an enormous number of instances to check if there are any exponential effects on performance. Lastly, we intend to compare how each generational garbage collector allocates and manages objects of different generations.

References

Beebee, W. & Rinard, M. (2001). An Implementation of Scoped Memory for Real-Time Java. EMSOFT.

Bertels, P., Wim, H., D'Hollander, E., & Stroobandt, D. (2009). Efficient memory management for hardware accelerated Java Virtual Machines. ACM Trans. Design Autom. Electr. Syst., vol. 14, pp. 48:1-48:18.

Craciunas, S. et al. (2008). A Compacting Real-Time Memory Management System. USENIX Annual Technical Conference.

Gherardi, L., Brugali, D., Comotti, D. (2012). A Java vs. C++ performance evaluation: a 3D modeling benchmark. Simulation, Modeling, and Programming for Autonomous Robots. Lecture Notes in Computer Science, vol. 7628.

Guyer, S. Z. & Lin, C. (2000). Optimizing the Use of High Performance Software Libraries. 13th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2000. Springer: Berlin, pp. 227–243.

Lengauer, P., Bitto, V., Mössenböck, H. & Weninger, M.(2017). A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. ACM: L'Aquila, Italy.

Nikishkov, G., Nikishkov, Y. & Savchenko, V. (2003). Comparison of C and Java Performance in Finite Element Computations. Computers and Structures, 81, pp. 2401-2408.

Qian, J. et al. (2016). Exploiting FIFO Scheduler to Improve Parallel Garbage Collection Performance. Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, vol. 51, n. 7. ACM: New York, USA, pp. 109 - 121.

Sestoft, P. (2010) Numeric performance in C, C# and Java. IT University of Copenhagen: Denmark. Retrieved from http://www.itu.dk/~sestoft/papers/numericperformance.pdf

Singer, J. (2003). JVM versus CLR: a comparative study. Proceedings of the 2nd International Symposium on Principles and Practice of Programming in Java, PPPJ 2003. Computer Science Press, Inc. New York: USA, pp. 167-169.

Ugawa, T., Abe, T. & Maeda, T. (2017). Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models. Proceedings of the ACM on Programming Languages, vol. 1. ACM: New York, USA, pp. 53:1 - 53:26.

Vassev, E. & Paquet, J. (2006). Aspects of Memory Management in Java and C++. Software Engineering Research and Practice.

Vivanco, R. & Pizzi, N. (2002). Computational Performance of Java and C++ in Processing Large Biomedical Datasets. Proceedings of the 2002 IEEE Canadian Conference on Electrical & Computer Engineering. Winnipeg: Canada, pp. 691 - 696.

Wescott, B. (2013). Useful Laws in The Every Computer Performance Book, Chapter 3. CreateSpace.

Zhang, J., Lee, J. & McKinley, P. K. (2005). Optimizing the Java Piped I/O Stream Library for Performance. In: Pugh B., Tseng CW. (eds) Languages and Compilers for Parallel Computing. LCPC. Lecture Notes in Computer Science, vol 2481. Springer: Berlin, pp. 233–248.