Enhancing the region model of real-time Java for large-scale systems

Pablo Basanta-Val, Marisol García-Valls, and Iria Estévez-Ayres

DREQUIEM LAB
DISTRIBUTED REAL TIME SYSTEMS AND MULTIMEDIA LABORATORY
Departamento Ingeniería de Telemática
Universidad Carlos III de Madrid
~http://www.it.uc3m.es/drequiem/
{pbasanta,mvalls,ayres}@it.uc3m.es

Abstract

The region-based memory model of The Real-time Specification for Java (RTSJ) is quite rigid, and it complicates the development of reusable predictable software for large-scale systems. In this paper, we propose an extension to the region model of the RTSJ called AGCMemory (Acyclic Garbage Collected Memory). This extension enables the destruction of floating garbage created during the execution of Java methods. The integration in the memory model of the RTSJ and the new required runtime barriers are addressed.

1. Introduction

As many other environments, large-scale systems can benefit from high-level languages in order to reduce application development cost. The complexity of such systems makes the Java language a good candidate for the following reasons: portability, automatic memory management, simplicity, and networking support. These features reduce the development time and help to produce applications that are easier to maintain. However, when we introduce the real-time constraint, the automatic memory management facility becomes a drawback; the garbage collector (GC), that automatically recycles the unused memory, introduces unpredictable pauses in program execution.

The problem of producing a predictable automatic memory management for Java does not have a perfect solution. The natural candidate is the real-time garbage collector technique, like the one described in [5]. This technique bounds the pauses in program execution, but its operation requires a priori information that is not easy to be calculated. Among such data, it requires the object allocation rate of each application and the maximum alive memory. Furthermore, the real-time garbage collector technique consumes extra memory and CPU; these extra resources are not always available in embedded systems. For this reason, current real-time Java specifications (The Real Time Specification for Java RTSJ [1] and the Real Time Core Extensions RTCE [2]) provide a lower level mechanism based on regions [10].

The predictability that the region model provides to the Java language makes it suitable for meeting time requirements at the nodes of critical large-scale real-time systems [6]. However, from the perspective of the programmer, there is a reduction of the automatic memory management benefits; the programmer has to collaborate in the automatic memory management.

The aim of this paper is to present a model that improves the portability of the regions of the RTSJ while keeping some of the advantages of the GC-based automatic memory management. In large-scale real-time systems, where multiple RTSJ-enabled nodes could be used, each node can execute a different version of libraries; each one has its own memory requirements. The proposed extension, AGCMemory, reduces the number of manual changes required to adapt existing libraries to execute following the region model.

The remaining of the paper is organized as follows: Section 2 reviews those research works that have influenced us in the design of this new type of region; Section 3 reviews the memory model of RTSJ explaining how we have integrated AGCMemory is this model; Section 4 exemplifies the benefits of AGCMemory by means of a simple periodic application; Section 5 deals with the internals of the implementation of AGCMemory, mainly its internal
structures and runtime barriers; Section 6 compares the performance of AGCMemory and scoped memory; and, finally, Section 7 presents our conclusions.

2. Related Work

One of the most active areas of research in real-time Java is the memory management. In this area, research is mainly directed towards the improvement of predictability and efficiency of the real-time garbage collectors [5] and the mitigation of problems that the region-based solution of RTSJ, scoped memory, posses (e.g.[4],[3],[9]). Our work falls into the second category.

Implicitly, scoped memory introduces two main problems: efficient validation of the run-time checks required to maintain the integrity of the model of scoped memory and the assignment of scopes to Java code. In [4], the efficient implementation of the runtime rules of RTSJ is addressed. The runtime checks required to validate the assignment rule, one of the most troublesome penalties introduced by the region model of RTSJ, is reduced from linear complexity to a constant time function using the display technique. The work presented in [3] deals with the automatic assignment of scoped memory instances to plain Java code. The main advantage of this technique is that it reduces the number of manual assignments. This is done in two phases. The first one is an off-line analysis, based in the escape technique, where the life of all objects is found out. In the second one, an automatic tool assigns the scopes to portions of the code using aspect programming.

RTCE [2] defines the stackable objects as a complementary mechanism to the region model. If an object is defined as stackable, the object will be created in the stack of thread, and it will be destroyed when the method ends; this is the same approach followed in Java for the local variables. RTCE avoids the problems of dangling pointers using static analysis of code that determines when a stackable object is referenced from outer objects.

Our proposed new subclass of scoped memory of RTSJ, AGCMemory, shares common ideas with the above mechanisms. As in [4], all extra run-time checks are performed in constant time. We share with [3] the idea of the definition of an automatic mechanism that assigns regions to code. However, our mechanism is executed on-line using runtime barriers instead of relying on an off-line analysis, as done in [2] and [3].

The use of run-time barriers performs a dynamic adaptation of the region structure to code. The escape analysis performed by our algorithm is simpler than the one in [3]; it is less powerful, but it reduces drastically the execution overhead. Eventually, our work may also be understood as an extension to stackable objects of RTCE in the context of RTSJ. In RTCE, the stackable objects allocated in the stack disappear after the end of the method where they are created, whereas objects created in an AGCMemory do not have this constraint.

3. Memory Management in the RTSJ

The memory model of RTSJ [1] is based in memory areas. Each memory area is related to a block of physical memory where Java objects are allocated by applications using the new or newInstance operators. In the RTSJ, there are tree types of memory areas:

-Heap Memory: It is the traditional heap of Java and there is a single instance in each virtual machine. Objects allocated in this memory area are garbage collected and its usage for real-time purposes requires a real-time garbage collector.

-Immortal memory: There is a single instance of immortal memory in each virtual machine and the objects that it contains can not be destroyed. In real-time environments this memory is typically used to store objects that have a life equals to the life of the virtual machine.

Scoped memory: Scoped memory instances enable the predictable allocation and de-allocation of objects. Unlike immortal memory and heap memory, the scoped memory instances are explicitly instantiated by the programmer which has to decide the amount of memory that each scoped memory instance has.

As heap memory, objects stored in a scoped memory instance may be reclaimed but instead of using a garbage collector mechanism, based in root scanning, scoped memory instances use an internal counter. When this counter reaches zero, all objects allocated in the scoped memory instance are destroyed.

In order to avoid dangling pointers to objects allocated in scoped memory, RTSJ imposes two rules: the assignment rule and the single parent rule. The verification of the rules is done at runtime by the virtual machine.
Integrating AGCMemory in the RTSJ

The scoped memory is an abstract class and it may not be directly instantiated. The programmer has to use one of its subclasses, LTMemory or VTMemory. The integration of AGCMemory in the class hierarchy of the RTSJ, as shown in figure 1, has been done extending the scoped memory. AGCMemory is a subclass of scoped memory and the same as all scoped memory subclasses, the AGCMemory is constrained to the assignment and single parent rule.

LTMemory and VTMemory classes differ in two main points. The first one is related to the allocation time: while in LTMemory it is bounded by a linear function, in VTMemory it is variable. The second one is the capacity of partial de-allocation of objects after the execution of methods: it is only supported by VTMemory; in an LTMemory instance, the management algorithm has to deal with the scope as a whole, it can not reclaim each object separately. Our approach, AGCMemory, tries to combine the good properties of LTMemory and VTMemory (linear bound of allocation time and support to the partial de-allocation of objects) leaving out their penalties.

4. Recycling floating garbage with AGCMemory

This section illustrates, using a simple example, how the recycling property of AGCMemory allows the destruction of the floating garbage created during the execution of methods, reducing the necessity of nested scopes.

In order to illustrate the problem of floating garbage, we have chosen a very simple application, PeriodicCounter. As it is shown in figure 2, PeriodicCounter has an infinite loop that increments a counter and prints out its value. In the constructor of PeriodicCounter (line 11), we associated an LTMemory instance to the run method of the thread. This involves that all objects created using new during the execution of the method run will be allocated in this LTMemory instance. As it can be seen in the code, it was initialized with 250 bytes of free memory.

```java
import javax.realtime.*;
public class PeriodicCounter extends RealtimeThread{
    public PeriodicCounter(){
        super(null, //Scheduling Parameters
            new PeriodicParameters(null,
                new RelativeTime(1000,0), //T
                new RelativeTime(50,0),   //C
                new RelativeTime(100,0), //D
                null,null);
            null,
            new LTMemory(250,250),
            null);
        start(); //starts current thread
    }//@constructor
    int counter=1;
    public void run(){
        do{ System.out.println(counter);
            counter++;
        }while(waitForNextPeriod());
    }//@run
    public static void main(String s[]){
        new HelloPeriodicCounter();
    }//@main
}
```

However, the application, as written, does not work in the JTime virtual machine [8]. Each time we print out the counter value, using System.out.println (line 17), the internal execution of method allocates 88 bytes in the LTMemory instance, and the third execution of loop will fail throwing an out-of-memory exception.

Typically, the way to eliminate these temporal objects created during the invocation of a method in RTSJ is to associate this invocation to another scope, called nested scope. The application of nested scope to the code of PeriodicCounter is shown in figure 3.

The nested scope is lt and it is initialized with 150 bytes of free memory. We also need the definition of an auxiliary class (impr), an inline runnable class, in order to associate the nested scope to the method println. The run method contains the logic (in our case the logic of the println method) that generates the temporal objects we want to eliminate. The invocation of enter changes the default allocation context to the nested scope, invokes the run method of impr and restores the default allocation context.
destroying the temporal objects created during the execution of the println method.

```java
import javax.realtime.*;
public class PeriodicCounter extends RealtimeThread{
    public PeriodicCounter(){
        super( null, //Scheduling Parameters
            new PeriodicParameters(null,
                new RelativeTime(1000,0), //T
                new RelativeTime(50,0), //C
                new RelativeTime(100,0), //D
                null,null);
        null,
            new LTMemory(250,250),
        null);
        int counter=1;
        public void run(){
            LTMemory lt=new LTMemory(150,150);
            Runnable impr=new Runnable(){
                public void run(){
                    System.out.println(counter);};
            lt.enter(impr); counter++;
            while(waitForNextPeriod());
        }//Run
        public static void main(String s){
            new HelloPeriodicCounter();
        }
    }
}
```

figure 3: Using nested scopes to eliminate floating garbage

At a glance, the use of an auxiliary scope comes with two new problems for the programmer: (1) the maximum size of the auxiliary scope has to be set (e.g. 150) and (2) it is necessary the definition of an auxiliary class (in the example, impr) containing the logic of those methods (System.out.println) whose temporal objects the programmer wants to destroy.

The set up of the size of auxiliary LTMemory presents a problem of portability: depending on the virtual machine and libraries we use, the println, for example, may create no temporal objects or it may create 1M of objects.

The definition of a new class that contains the code that allocates objects make code hard to read because this auxiliary class does not come as a modeling prerequisite but as an imposition of the memory model of the RTSJ.

Using AGCMemory, we are able to eliminate temporal objects created in the execution without using an auxiliary nested scoped. Thus, in the example of figure 2, only one line has to be changed. Instead of assigning a LTMemory (line 11: new LTMemory(250,250));, we have to instantiate an AGCMemory (new AGCMemory(250,250)); no nested LTMemory of figure 3 is required.

Finally, the example is simple and helps us to understand the benefits of AGCMemory but it does not give us information about the importance of the problem of floating garbage in Java. In current J2SE library classes, this is an important problem: more than 50% of the class methods create objects during their invocation [7].

5. AGCMemory: Internals

This section deals with the internals of AGCMemory. First of all, we present the properties of AGCMemory that concerns to the implementation of its automatic memory management. The section continues by describing its static properties, i.e. the internal data structures. Finally, we describe the runtime barriers that modify this internal data structures.

Main features

No-shared memory

The AGCMemory is conceived as a private memory for the execution of a thread. If we try to share it among several threads, only one thread will be able use it at the same time. Whereas the memory is in use, the threads that try to use it, invoking enter, will get an exception, indicating that the memory is being used. This simplification will help us to define a constant time mechanism that detects dead objects and collect them during program execution.

Predictable memory de-allocation

The de-allocation of objects allocated in an AGCMemory was designed to be time predictable and this fact has implications in the automatic memory management algorithm. We should not use complex algorithms based in root scanning because its execution after the invocation of every method would damage the performance of the virtual machine. Instead of using these general mechanisms, we have to use others, less precise but easier to predict like the one described below.

Memory model and data structure

As shown in figure 4, each AGCMemory has a chunk of physical memory. This physical memory is addressed as linear memory space. The objects are
allocated in this memory in ascending memory addresses. The free memory pointer, `free_mem_ptr`, points to the memory position where the next object will be allocated.

![Figure 4: Internals of the AGCMemory](image)

Besides, the AGCMemory contains a complementary structure, the `agc_stack`. This structure is related to the memory management algorithm and it contains the information that will enable us the partial recycling of objects. Each entry of the `agc_stack` contains two elements, the method pointer, `method_ptr`, and the escape pointer, `scape_ptr`. The method pointer keeps information about the set of objects that are created during the invocation of a method. The escape pointer decides whether the objects created during the execution of a method may be recycled or not.

**Runtime barriers**

The automatic memory management of the AGCMemory is based in runtime barriers that are executed before and after each Java method and in global reference assignments. The execution of these runtime barriers combined with the information of the `agc_stack` destroys the temporal objects created during the execution of the methods.

**pre_invocation_barrier**

The pre-invocation barrier is executed just before the invocation of a Java method. It pushes a new entry in the `agc_stack`. Both values the entry are initialized with the same value, `free_mem_ptr`. That is:

\[
\begin{align*}
\text{top} \rightarrow \text{scape_ptr} &= \text{free_mem_ptr} \\
\text{top} \rightarrow \text{method_ptr} &= \text{free_mem_ptr}
\end{align*}
\]

**post_invocation_barrier**

The post-invocation barrier is executed just after the invocation of a Java method. This barrier performs two actions. First, it pops an entry from the `agc_stack` and after it decides whether or not to recycle objects allocated during the method execution.

The test performed to detect whether the objects may be destroyed consists in the verification of the following condition:

\[
\text{top} \rightarrow \text{scape_ptr} \geq \text{top} \rightarrow \text{method_ptr}
\]

When the condition is true, all objects allocated in the range \([\text{top} \rightarrow \text{method_ptr}, \text{free_mem_ptr}]\) are destroyed and the value of `free_mem_ptr` is set to `top \rightarrow \text{method_ptr}`. When condition is not fulfilled, the responsibility of the destruction of the objects is propagated to the parent method; assigning the value of `top-1 \rightarrow \text{scape_ptr}` to `top \rightarrow \text{scape_ptr}`.

Notice that the de-allocation of objects never fragments the memory. When any of the objects created during the invocation to a method is referenced by an object created in an outer method, all objects are maintained.

**assignment_barrier**

Additionally, there is a barrier that updates the value of `top \rightarrow \text{scape_pointer}`. This barrier is executed each time we assign a reference to an object. The purpose of the barrier is to detect whether the objects created during the execution of the method can be destroyed when the method ends. Given a reference attribute of an object, `attrib`, that try to refer to another object, `ref`, the assignment `attrib=ref` executes the following barrier:

```java
if (memArea(attrib)==memArea(ref))
&& (memArea(attrib) instanceof AGCMemory)
&& (ref \geq attrib)
\text{top} \rightarrow \text{scape_ptr} = \text{min} \{ \text{top} \rightarrow \text{scape_ptr}, \text{attrib} \}
```

Notice escape analysis employed is pessimistic. Once a reference escapes the method closure, the algorithm considers that all objects allocated in the method also escapes. So, the possibility of that the reference assigned to the attribute of object is destroyed before the method ends is not consider by our algorithm.

6. Discussion

AGCMemory and the traditional scoped memory may be compared in terms of two parameters: efficiency in the use of memory and overhead introduced in runtime.
In terms of memory efficiency, the AGCMemory looks, in general, more efficient than traditional scoped memory. The elimination of floating garbage performed by AGCMemory allows collecting dead objects before leaving (exit method) the scope, reducing the amount of memory needed for an application when compared to the nested scope mechanism.

In terms of introduced overhead it is not clear whether the AGCMemory is or not more efficient than traditional scoped memory. There is a strong dependency on the type of application and on the underlying execution environment. On the one hand, it seems more efficient because it reduces the necessity of using nested scoped memory instances and, implicitly, the cost of runtime checks of the assignment rule (with a linear dependency with the nesting level). On the other hand, when the validation of rules is optimized using constant time barriers [4], the assumption is not true and AGCMemory may introduce appreciable overheads.

7. Conclusions

The current types of region of RTSJ obligate us to choice between a bounded time of allocation, using LTMemory instances, or a reduced amount of memory used by the applications, using VTMemory instances. In this paper, we presented a new type of region for the RTSJ, the AGCMemory that addresses the problem of providing these two desired characteristics together. The automatic memory management of the AGCMemory is able to eliminate floating garbage created during the invocation of Java methods using constant time barriers. It also reduces the necessity of nested scopes, being more suitable for large-scale systems. Our future work will include the efficient implementation of the AGCMemory in order to quantify the memory consumption reduction and the overhead introduced by AGCMemory.

References