AGCMemory: A new Real-Time Java Region Type for Automatic Floating Garbage Recycling

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 transparent destruction of floating garbage created during the execution of Java methods. The integration within the memory model of the RTSJ and its automatic memory management algorithm, based on run-time barriers, are described.

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, 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 [6]. This technique bounds the pauses in program execution, but its operation requires a priori information that is not always easy to calculate. 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 [11].

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 [7]. However, from the perspective of the programmer, there is a reduction of the automatic memory management benefits; the programmer has to collaborate in such a process.

The aim of this paper is to recover some of portability provided by general collectors in the context of regions. The idea is to produce an enhanced region that detects unused objects and recycles them when they become unreachable. The automatic memory management algorithm of this new type of region will not be as complex as general garbage collectors.

In large-scale real-time systems, where multiple RTSJ-enabled nodes are used, and each node executes a different version of the RTSJ libraries, the new type of region (called AGCMemory) may reduce the number of manual changes required to adapt existing libraries to the region model of the RTSJ.

The rest 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 presents the memory model of RTSJ explaining how we have integrated AGCMemory is this model; Section 4 exemplifies the benefits of AGCMemory in the context of floating garbage; Section 5 deals with the internals of the

implementation of AGCMemory, mainly the structures and run-time barriers involved in its automatic memory management algorithm; Section 6 draws some 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 [6] and the mitigation of problems that the region-based solution of RTSJ, scoped memory, posses (e.g.[4],[3],[5],[10]). 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 run-time rules of RTSJ is addressed. The run-time 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 the thread. Also, it will be destroyed when the method ends, using 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 run-time 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 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 have to be destroyed when the method ends, whereas in an AGCMemory region they do not have to.

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 three 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 equal 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.

Besides, in order to avoid dangling pointers to objects allocated in scoped memory, RTSJ imposes two rules: the *assignment rule* and the *single parent rule*. These rules are verified by virtual machine using run-time barriers.

Integrating AGCMemory within the RTSJ

The scoped memory is an abstract class that 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; like all scoped memory subclasses, the AGCMemory is constrained to the assignment and single parent rule.

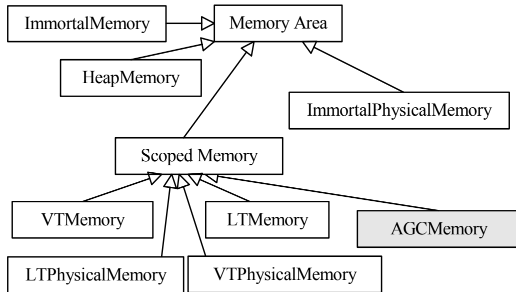

Figure 1: AGCMemory within the class hierarchy of RTSJ

In some sense, AGCMemory combines the advantages of both LTMemory and VTMemory. On one hand, the use of LTMemory guarantees that the allocation time for objects is bounded by a linear function whereas in VTMemory this is not guaranteed. And on the other hand, VTMemory may support the partial de-allocation of objects during the instantiation of new objects, improving the reusability of memory. In the AGCMemory, the allocation time is bounded by a linear function, and it supports the partial de-allocation of objects after the invocation of Java methods.

4. Recycling floating garbage with AGCMemory

Using a simple example, this section shows how the recycling property of AGCMemory enables the destruction of the floating garbage created during the execution of Java 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 shown in figure 2, PeriodicCounter has an infinite loop that increments a counter and prints out its value.

In the PeriodicCounter constructor (line 11), we associated an LTMemory instance to the `run` method of the thread. This means that all objects created using `new` during the execution of the method `run` will be allocated in this memory area instance.

```
01: import javax.realtime.*;
02: public class PeriodicCounter extends RealtimeThread{
03:    public PeriodicCounter(){
04:          super( null, //Schedulling Parameters
05:                 new PeriodicParameters(null,
06:                 new RelativeTime(1000,0),//T
07:                 new RelativeTime(50,0),  //C
08:                 new  RelativeTime(100,0),//D
09:                 null,null);
10:                 null,
11:                 new LTMemory(250,250),
12:                 null);
13:          start(); //starts thread
14:          }//@constructor
15:    int counter=1;
16:    public void run(){
17:          do{System.out.println(counter);
18:             counter++;}while(waitForNextPeriod());
19:          }//@run
20:    public static void main(String s[]){
21:          new HelloPeriodicCounter();
22:          }
23:    }
```

Figure 2: Full code of PeriodicCounter application

However, the application, as it is written, does not work in all virtual machines. In JTime virtual machine [9], it will throw an out-of-memory exception. Each time we print out the counter value, using `System.out.println`, the method allocates 88 bytes within the `LTMemory (250,250)` instance in order to convert the int value into a string. During the third invocation, the lack of free memory will cause an exception to be thrown.

In RTSJ, the way to eliminate these temporal objects created during the invocation of a method is to use nested scopes.

Figure 3 shows how to use nested scopes to eliminate the temporal objects created during the invocation of `println`. The nested scope is `lt` and it manages 150 bytes of memory. `Impr` is a runnable object and it contains the code, in our case `println` method, whose temporal objects we want to eliminate. The mechanism to bind the memory `lt` and the logic `impr` of `run` is the `enter` method. When `lt.enter(impr)` is invoked, the virtual machine performs three actions: it changes the default allocation context of objects to `lt`; it executes `impr.run()`, allocating the objects in `lt`; and it destroys objects allocated in `lt` memory, restoring the allocation context.

```
01: import javax.realtime.*;
02: public class PeriodicCounter extends RealtimeThread{
03:    public PeriodicCounter(){
04:           super( null, //Schedulling Parameters
05:                  new PeriodicParameters(null,
06:                  new RelativeTime(1000,0), //T
07:                  new RelativeTime(50,0),   //C
08:                  new  RelativeTime(100,0), //D
09:                  null,null);
10:                  null,
11:                  new LTMemory(250,250),
12:                  null);
13:                  start();
14:            }
15:    int counter=1;
16:    public void run(){
17:                  do{ lt.enter(impr);
18:                  counter++;}while(waitForNextPeriod());
19:           }
20:    LTMemory lt=new LTMemory(150,150);
21:    Runnable impr=new Runnable(){
22:                  public void run(){
23:                  System.out.println(counter);}};
24:    public static void main(String s){
25:                  new HelloPeriodicCounter();
26:           }
27:    }
```

Figure 3: Using nested scopes to eliminate floating garbage

Figure 4 shows the memory profile of the two mechanisms. Whereas the use of a non-nested scope is not able to recycle the objects created during the `println` invocation, crashing during the third execution of the loop, the nested scope recycles the 88 bytes required to convert the `int` value into a string.

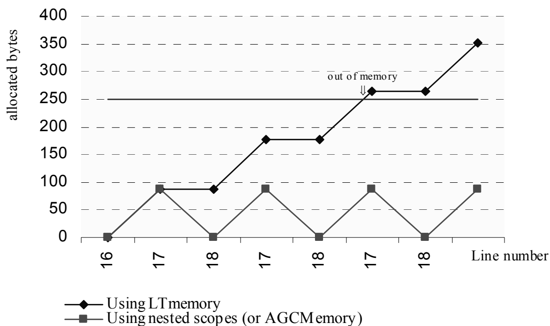

Figure 4: Allocated memory profile in PeriodicCounter

However, the use of an auxiliary scope comes with several problems. Firstly, the use of runnable objects difficult the maintenance of programs. These new objects do not come as a result of a requirement in the modeling of applications but as an aspect of memory management. Secondly, we have to analyze which methods of the API create objects during its invocation. This may be complex because depending on the implementation, a method may create temporal objects during its invocation or not. Eventually, we have to dimension (e.g. 150 bytes) the size of each scope. Once again, this is complex because the number of objects allocated during the execution of a method may depend on the platform and on the invocation parameters.

The AGCMemory was designed to mitigate these problems; it hides some of these complexities to the programmer. The AGCMemory is able to collect floating objects created during the invocation of Java methods after its execution.

In the PeriodicCounter example, we are able to eliminate the temporal objects created during the invocation of `println` without requiring the use of a nested scope. The only requirement is that instead of using an LTMemory, we use and AGCMemory region. Therefore, the code of figure 2 only requires one change in line 11: the replacement of `new LTMemory(250, 250)` statement with `new AGCMemory (250,250)`. Once changed, the memory profile of the application will be same that we obtained for the nested scopes, shown in figure 4.

Eventually, the example is simple helping us to understand the benefits of AGCMemory; however, it does not give us information about the importance of the problem of floating garbage in Java. In Java, this is a relevant problem. In current J2SE library classes, more than 50% of the class methods may create objects during their invocation [8].

5. Supporting AGCMemory

In the previous section we have illustrated how AGCMemory may reduce the number of manual adaptations on code; however, we have not dealt with the implementation details. In this section we will explore the utilization of a non-shared and imprecise algorithm for such purpose. Similarly to [4] and [5], the algorithm has been supported using run-time barriers.

Algorithm features

In order to simply the AGCMemory support, two main restrictions are introduced by our algorithm.

The first has been placed in the AGCMemory shareability. An AGCMemory may not be concurrently used by several threads; only one thread may allocate objects on it. Consequently, when a thread enters (e.g. using `enter(Runnable r)`) an already entered AGCMemory instance, it gets a memory in use exception.

The second limitation is in the garbage detection. For garbage collector detection and elimination, the proposed algorithm treats all objects created during method execution as if they were a single object. That is, the algorithm eliminates all objects created during the method invocation or neither of them; partial object elimination is not supported.

Memory model and data structure

As shown in Figure 5, each AGCMemory has a chunk of physical memory. This physical memory is addressed as linear memory space. The objects are allocated in it in increasing addresses number. The free memory pointer, `free_mem_ptr`, points to the position where the next object will be allocated.

Besides, the AGCMemory contains a complementary structure, the `agc_stack`. This structure is related to the memory management algorithm; it contains the information that will allow 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.

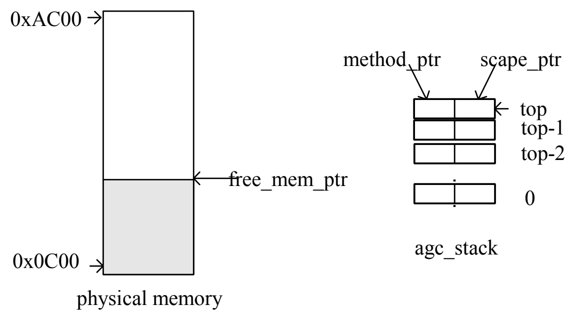

Figure 5: Internals of the AGCMemory

Run-time barriers

As said before, automatic memory management of the AGCMemory is based on run-time barriers. In our case, the barriers are executed before and after each Java method and in each reference assignment. The execution of these run-time barriers combined with the information of the `agc_stack` destroys the temporal objects created during the execution of the Java 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 entry values are initialized with the same value: `free_mem_ptr`. That is:

```
top→scape_ptr=free_mem_ptr;
top→method_ptr=free_mem_ptr;
```

The purpose of the barrier is to be able to detect which objects are created during the execution of a method.

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`. 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 of the verification of the following condition:

```
top→scape_ptr ≥ top→method_ptr
```

When the condition is true, all objects allocated in the range `[top→method_ptr, free_mem_ptr]` are destroyed and the value of `free_mem_ptr` is set to `top→method_ptr`.

When this condition is not fulfilled, the responsibility of the destruction of the objects is propagated to the parent method performing this assignment `top-1→scape_ptr = min{top-1→scape_ptr, top→scape_ptr}`

assignment_barrier

Additionally, there is a barrier that updates the value of `top→scape_pointer`. This barrier is executed before reference assignments.

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`, trying to refer to another object, `ref`, the assignment `attrib=ref` executes the following barrier:

```
if(memArea(attrib)==memArea(ref))
&& (memArea(attrib) instance of AGCMemory)&&
(ref≥attrib))

top→scape_ptr=min{top→scape_ptr, attrib}
```

Besides, the assignment barrier is executed each time a method returns a reference. In this case, the executed barrier is the following:

```
top→scape_ptr=min{top→scape_ptr,top→method_p
tr-1}
```

As main conclusion we may say that introduced restrictions have enabled the implementation of a lightweight approach. Proposed solution provides us two main advantages: the first is that the memory structure used to detect dead objects may be a stack-based one and the second is that run-time barriers checks complexity may be bounded by a constant time function.

6. Conclusions

Current types of regions in the RTSJ force us to choice between a bounded time allocation, using LTMemory instances, or an efficient memory management, using VTMemory. The combination of both features is not directly supported. The predictable elimination of the floating garbage requires collaboration from the programmer who has to insert nested scopes in application code.

In order to provide a more portable region model we have defined a new type of region, AGCMemory. This new type offers an alternative mechanism where the detection and the floating garbage elimination have been hidden to the programmer. As consequence of this, the necessity of having nested scopes and manual configuration has been reduced. This functionality is now supported by the virtual machine.

References

[1] RTJEG. "The Real Time Specification for Java", Addison Wesley, 2000. Available at http://www.rtj.org

[2] J-Consortium Inc. "Core Real Time Extensions for the Java Platform" September 2000 Available at http://www.j-consortium.org

[3] Deters, M. "Dynamic Assignment of Scoped Memory to Regions in Translation of Java to Real-Time Java" M.S Thesis. Washington University. Department of Computer Science, May 2003

[4] Corsaro, A; Cytron, R. K. "Efficient Memory Reference checks for real-time java" In proceedings of the ACM Conference on Languages, Compilers and Tools for Embedded Systems (LCTES03). San Diego, California June 2003. pp 51-58

[5] Higuera-Toledano, M.T. et al.; Memory Management for Real-Time Java: An Efficient Solution using Hardware support. Real-Time Systems 26(1): 63-87 (2004).

[6] Bacon, David F.; Cheng, P.; Rajan, V.T. "Metronome: A Simpler Approach to Garbage Collection in Real Time Systems". First Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES03). Catania, Sicily, November 2003. LNCS vol. 2889, pp. 166-178

[7] Sharp, D.C.; Pla, E.; Lueke, K.R. "Evaluating mission critical large-scale embedded systems performance in Real-time Java." In proc. of 24th IEEE Real Time Systems Symposium (RTSS03). December 2003. pp. 362-365

[8] Dibble, P. "Non-Allocating Methods" Draft 1 for discussion, September 2004. Available at http://www.rtsj.org

[9] Timesys Corp. "JTime virtual machine". Available for downloading at http://www.timesys.com

[10] Basanta-Val, P.; García-Valls, M.; Estevez-Ayres, I. "Towards the Integration of Scoped Memory in Distributed Real-Time Java" In proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC05). Seattle, Washington. May 2005.

[11] Toefte, M.; Birkedal, L.; Elsman, M.; Hallenberg, N. "A Retrospective on Region-Based Memory Management" Higher-Order and Symbolic Computation Journal, volume 17, number 3. September 2004 pp. 245-265