

Eliminating Garbage Collection for Embedded Real-Time Software

Nader Mohamed and Jameela Al-Jaroodi
Dependable Systems Middleware (DeSyM) Research Lab.
Electrical and Computer Engineering Department
Stevens Institute of Technology
Hoboken, NJ 07030
{nmohamed,jaljaroo}@stevens.edu

Abstract

One of the problems with Java for real-time and embedded real-time systems is the unpredictable behavior of garbage collection (GC). GC introduces unexpected load and causes undesirable delays for real-time applications. In this paper, we propose a technique that reduces and bounds the memory requirements for real-time and embedded Java programs. This technique eliminates the need for GC and allows for a more deterministic execution behavior and efficient utilization of the available memory. A number of benchmark tests are used to evaluate this technique in PERC, NewMonics' real-time JVM, and Sun's JVM. The results show that GC can be eliminated and an application's execution time decreases and becomes more deterministic. In addition, we briefly introduce a framework to automate the technique.

Keywords: Embedded Real-Time Applications, Garbage Collection, Java

1. Introduction

For Java to support real-time applications, a number of issues must be taken into consideration [3][4]. One important aspect to be addressed is the need for efficient and deterministic dynamic memory management. Regular Java implementations allow users to indefinitely instantiate objects and rely on automatic garbage collection (GC) [9] to clean-up unused objects (see Figure 1). Moreover, real-time applications that generate a large number of objects can fill the available memory frequently. Therefore, GC is frequently executed by the JVM to free the memory. This means that the behavior (e.g. response time) of the application becomes non-

deterministic because GC will start at any point in time and it is generally not preemptable. One approach to limit the use of GC is to statically pre-allocate the memory needed by the application. However, this reduces the utilization of memory and restricts the number of tasks that can execute, which may reduce CPU utilization.

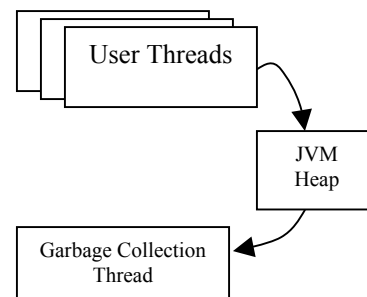


Figure 1. Current JVM memory management model, where user threads create objects and a GC thread collects unused objects to free the heap. The GC thread executes with non-deterministic timing behavior.

Real-time applications such as application-level routers or on-line networked multimedia applications may create many objects from the same class. Each network packet or MPEG frame may be represented as an object, for example. Although these objects have a short lifetime, they will stay in the heap for some time until GC collects them with non-deterministic timing behavior, which may cause unacceptable delays. The object-reuse technique can eliminate the need for GC in this type of situations, thus providing a deterministic application behavior and more efficient memory usage.

In this paper, we propose a technique to eliminate the need for GC, while maintaining dynamic memory allocation. The technique is

based on two basic ideas. (1) Garbage collection only executes if the heap utilization reaches a preset threshold; why not confine memory usage to this limit. (2) Object-reuse and object pools are currently used in some contexts to reduce the memory usage for database and server applications; why not for real-time and embedded applications. By combining these two concepts, the undeterministic behavior of GC can be avoided by eliminating the need for GC altogether. This is achieved by combining the object-reuse techniques with the restriction on the memory size used by the application at the same time.

The rest of this paper first discusses background information and related work on real-time Java (RTJ) and RTJ memory management in Section 2. Section 3 describes the proposed solution and its implementation. Section 4 describes the evaluation of the solution and presents the results. In Section 5, we introduce a framework for automating the object-reuse technique. Section 6 concludes the study with some remarks about current and future work.

2. Background and Related Work

In the late nineties, researchers began investigating the suitability of Java for Real-time applications. In 1999, the Requirements Group for Real-time Extensions for the Java Platform, which includes the NIST and over fifty other organizations from academia and the industry, issued a report for RTJ requirements [4]. However, to make Java suitable for real-time applications, a number of issues still need to be addressed [4][3][11]. These include predictable GC, faster execution, and better real-time programming APIs. In addition, portability, dynamic adaptability and fault tolerance must also be considered [8].

The Real-Time for Java Expert Group was chartered under the Java Community Process to produce a specification for additions to the Java platform to enable Java programs to be used for real-time applications. The resulting Real-Time Specifications for Java (RTSJ) [10] provide clear guidelines for RTJ systems, but allows for flexibility in the implementation.

In the non-real-time JVM, memory allocation is done through a heap of specified size. Users can create objects, which are placed in that heap. When the heap reaches a certain threshold, GC is invoked to remove the unused objects from the heap to free up some memory and defragment the heap. In many cases, GC requires a long time to complete, thus delaying all other tasks.

In RTJ, GC (recently referred to as memory recycling) is identified to be a low priority thread that executes in the background. In reality, however, it has to halt the running application to correctly perform its tasks. This causes a problem given the tight bounds on execution time and memory consumption. A study of the traditional GC algorithms can be found in [6]. Although a number of techniques are used to address this problem, they all rely on the existence of the GC in some form. The technique we introduce combines some of the advantages of these techniques to eliminate GC and, at the same time, not limit the memory utilization.

3. The Object-Reuse Technique

In this section, we discuss an approach for efficient and deterministic memory management for RTJ. This approach combines some of the good features in the currently available methods (as described in Section 2). The solution is based on providing mechanisms to reuse existing unneeded objects instead of instantiating new ones. The first implementation provides the developer with two primitives to create and destroy objects to minimize the memory requirements of the application. However, this implementation will introduce some overhead on the user when using this method. Although the addition of the primitives seems to add some burden on the user, they are used as a proof-of-concept. Later, a framework for automating this technique will be given outlining the different issues required to make the solution transparent to the user.

As explained earlier, many real-time applications repeatedly create multiple instances of an object to be used temporarily. These objects remain in the heap for some time until GC cleans the heap. However, this means that GC will repeatedly be invoked and delay all other threads.

As a result, it becomes difficult to estimate realistic time bounds on the execution, which is essential to effectively schedule real-time tasks. The primitives we provide allow the user to create objects as needed and destroy them when done. This will free the space used by these objects to be reused by new ones (see Figure 2).

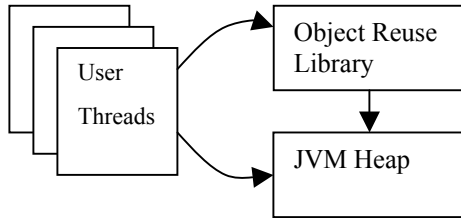


Figure 2. Proposed model where user threads use object-reuse primitives to create new objects and to declare unwanted objects. GC thread can be disabled to have deterministic timing behavior.

Using these primitives, in place of the usual object instantiation methods available in Java, leads to some desirable effects:

- 1 Eliminating (or in the worst case minimizing) the need for executing GC, thus removing the non-deterministic factor in tasks' execution time and response time.
- 2 Providing a more efficient use of available memory through reuse. This is especially important for systems with very limited memory (e.g. embedded systems) and for systems that generate many intermediate and temporary objects for processing.
- 3 Reducing the cost of instantiating objects since the total number of new objects to be instantiated will be much smaller. Moreover, the cost of creating and destroying objects for reuse is significantly smaller than the cost of a new object instantiation.

The effect of using the reuse solution is illustrated in Figures 3 and 4, which show the timeline of execution of an application in both cases. Figure 3 shows the response time for an application with regular object instantiation and GC operations. Figure 4 shows the response time of the same application when the new primitives are used and GC is not invoked.

A prototype was built to implement and evaluate the technique; a class called OR (for Object Reuse) is designed to provide two primitives (*create* and *destroy*). In this class two data structures are maintained:

- 1 `activeObject` is used to maintain a reference for each object that is currently in use.
- 2 `deletedObject` is used to maintain a reference for each destroyed object.

When the *create* primitive is executed, it will first check if the object to be created is in `deletedObject`. If it is there, the object reference is moved to `activeObject` and the object is re-initialized and returned to the user. Otherwise, a new object is instantiated for the user and its reference is added to `activeObject`. The *destroy* primitive will simply move the object's reference from `activeObject` to `deletedObject`. To successfully utilize these primitives the user will need to modify the object constructors to allow for the re-initialization of reused objects. Figures 5 and 6 show an example using the *create* and *destroy* primitives.



Figure 3. Current memory management model: GC thread works with non-deterministic behavior.

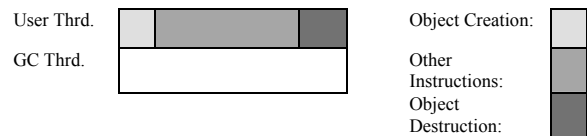


Figure 4. Timing behavior for proposed deterministic model.

```

UserRequest request;
While( there are more requests ) {
    // create new user Request object
    request = new userRequest(parameters);
    // receive user request
    receive(request);
    ...
    // handling user request
    ... }

```

Figure 5. User program, where each `userRequest` object lifetime is one iteration of the while loop. However, the `userRequest` object destruction is done by GC with non-deterministic timing behavior.

```

while( there are more requests )
{ // Create new user Request object using the
  // suggested library
  request = OR.create("userRequest",parameters);
  // receive user request
  receive(request);
  ...
  // Handling user request
  ...
  // object destruction
  OR.destroy(request);
}

```

Figure 6. User program where the suggested model is used to construct a `userRequest` object at beginning of the loop and destruct the request object at the end of the loop. GC thread can be eliminated.

4. Evaluation

To measure the performance of the object-reuse technique, a number of benchmark examples are used. The first set of experiments used a micro-benchmark to measure individual components such as the response time of tasks with and without object-reuse. In addition, an application benchmark, Java Object Router (JOR) [7] is used to show the overall performance. Java SDK 1.4 and PERC (real-time JVM) were used.

4.1. Micro-benchmark results

The micro benchmark simulates an image processor, where the input is a series of images to be processed and sent out. This requires an object for every incoming image, which is not needed after processing the image. Figure 7 shows code segments for image creation and processing using regular Java (ImageGC, left) and using the object-reuse technique using the OR class (ImageOR, right). Heap size was fixed to 8MB and the image object is binary of size 2000x2000 bits requiring 500,000 bytes of memory. Table 1 shows some measurements for two versions of GC (full and incremental) using SUN JVM. From the results, the object-reuse code takes shorter time in both settings. Generally, the incremental GC version has higher overhead than full GC, but the advantage is that it works for shorter periods every time it is invoked, thus reducing the delays on other tasks. When PERC[12] was used Table 2

with active GC, the time for the regular program was higher than the object-reuse program. With GC disabled, the regular program failed due to insufficient memory, but the object-reuse version continued normally. This shows that the reuse technique allows the program to continue executing within the limited memory available.

```

// ImageGC: Loop with new object instantiation
For(i=0;i<50;i++)
{
  // create new object instance
  image = new ImageScan();
  // get an image
  camera.getImage(image);
  // process the image object
  processImage(image);
}

```

```

// ImageOR: Loop with object-reuse
for(i=0;i<50;i++)
{
  // use object reuse library to create new object
  image = (ImageScan)
  OR.create("ImageScan");
  // get an image
  camera.getImage(image);
  // process the image
  processImage(image);
  // destroy the image object
  OR.destroy(image);
}

```

Figure 7. Code segments using regular Java and using the object-reuse technique.

Table 1. Overall measurements of loop timings and GC activities on SUN JVM.

Prog.	Normal Garbage Collector			Incremental Garbage Collector		
	Program loop time in sec.	Number of GC activities	Total GC time in sec.	Prog. loop time in sec.	Number of GC activities	Total GC time in sec.
New object creation	6.169	51	0.171	6.740	258	0.583
Object-reuse	5.107	0	0.000	5.107	0	0.000

Table 2. Overall measurements of loop timings on PERC.

Program	With GC (time in sec)	Disabling GC (time in sec)
New object creation	8.773	Out of memory Error
Object-reuse	7.521	7.661

The second experiment measures the execution time of the iterations in the programs one at a time. A description and full results of this experiments can be found in [1]. Generally, the PERC JVM (avg. 180ms per iteration) is slower than the SUN JVM (avg. 140ms per iteration). Moreover, the first iteration in the object-reuse program is very high because object instantiation is performed in this iteration, but it is not required in the next iterations. The variance in execution time for the regular program is very high (around 50ms) because of the GC activities. Nevertheless, in the object-reuse program, the variance is much smaller (around 20ms on SUN JVM and 10ms for PERC with disabled GC) because the heap is better utilized. Other traces on the SUN JVM with full and incremental GC were conducted to give an idea about the non-deterministic GC behavior [1]. The execution time with full GC is 6.17 seconds of which 0.171 seconds for full GC, while it was 6.6 seconds of which 0.583 for incremental GC.

4.2. Application Benchmark

JOR [7] is an application level router that routes Java objects among distributed Java applications on a network based on object type, contents, or object source. Since JOR handles objects to be routed among distributed applications, new objects are created for every received request. These objects are only needed during the service time of the request. The aim is to have deterministic routing process delay under certain workloads. To setup this experiment, the JOR router thread was modified slightly such that a time stamp is registered as soon as an object is received and another is registered as soon as the object is ready to be transferred. The difference between the timers gives the processing time of the object packet. The time registered avoids counting for any communication costs to clearly highlight the processing costs only. In addition, the server was set to send object packets at a relatively slow rate to avoid filling the router queue; thus, the time registered will not be affected by queuing delays. Two versions of the router threads were used, one is the original router with active GC and the other includes the object-reuse primitives. The results in Table 3 show that the variations in execution times in the regular router are very high (3 to 30ms) even

though the process is the same for all packets. This is contributed to GC activities that are triggered non-deterministically during the execution. On the other hand, the execution time with the object-reuse technique has much less variation (2 to 6ms). Moreover, the average execution time of the regular Java program is 5.78ms per packet, but drops to 2.41ms for the object-reuse technique. This shows that most of the activities in the regular router are due to the object instantiation and GC activities. In the object-reuse router, the cost of instantiation is minimized and GC cost is eliminated. The performance gain is an additional advantage to the object-reuse technique along with the elimination of GC activities.

As stated earlier, this approach is most suitable for service type applications that repeatedly generate objects to satisfy some request, which requires large heap space. However, these objects are not needed as soon as the request they service is completed. In this case, the object-reuse technique becomes very useful.

Table 3. The results of JOR experiment.

	Router with GC	Router with OR
Number of Packets processed	100	100
Average time per packet	5.78	2.41
Minimum packet processing time	3	2
Maximum packet processing time	30	6
Difference (Max – Min)	27	4
Standard deviation	6.07	0.66

5. Automating the Object Reuse Technique

Using the provided class primitives may be considered an added complication from the user's point-of-view. Therefore, a framework to automate the technique is described here. The automation can be done at different levels such as a preprocessor (Java to Java), as part of the compiler, or as a post-compiler (Bytecode to Bytecode). Figure 8 shows an example of a pre-compiler automation tool. In this case, the tool should be able to analyze the Java code and add the necessary primitives for object-reuse.

Ideally, to automatically add object-reuse to an application, we need to know exactly how and

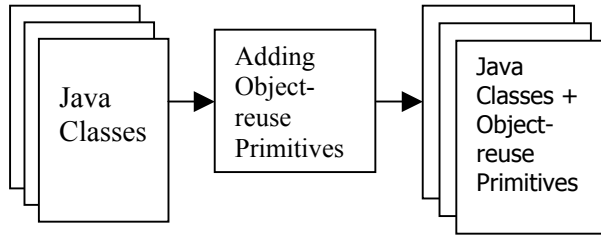


Figure 8. Off-line process to automatically add object-reuse primitives to user classes.

where each object is used. This will allow the system to replace every object instantiation by the *create* primitive and add the necessary *destroy* primitives for the unused objects. To add the *destroy* primitives, we must identify the last reference to an object by analyzing the code; however, this is a very difficult task.

Theorem 1. Identifying the last reference to an object is an undecidable problem.

Proof: To prove the theorem we will rely on a commonly known undecidable problem, the halting problem. The question is: given a program and a certain instruction (in this case a reference to an object), is it possible to determine if the program will reach that instruction? Let the instruction that contains the last reference to an active object be I_o . Using the halting problem, we can reformulate our problem by creating a Turing machine that adds the I_o instruction at the end of the program of the halting problem, thus the question becomes whether this instruction will be reached or not (will the new Turing machine halt?). If we can answer this question, we will solve the halting problem. This leads to the conclusion that identifying the last reference to the object is also undecidable.

□

Based on Theorem 1, we need an approximation mechanism to try to find these references. Some tools, such as a profiler, to help identify program and object usage characteristics are needed before trying to add the primitives. The profiler will execute the code in all possible cases and create a trace of the program flow and the object usage. For each object instantiated, the profiler will register the time and location of the object's instantiation and of all future references to that object.

Using the flow information, a flow graph can be constructed such that each node represents the largest possible contiguous block of instructions (has exactly one entry and one exit point). For each object, the instantiation and reference points must be identified in the flow. By taking the different possibilities of object references in the flow graph, we can find the last nodes where that object was referenced, thus the *destroy* primitive can be added at the end of that block. Nevertheless, care must be taken when multiple object instantiations occur during the different cases of execution. Here the profiler must compare objects not only by name, but also by the class they are from and the location of instantiation. Objects in this form are considered the same object and should be treated as such. However, this technique relies on the availability of a full flow graph of the program such that all possible cases are considered and no object references are overlooked. Since tracking all possibilities is a very tedious task (may have huge number of possibilities), it may be feasible to allow the user to identify the important cases of execution for the profiler.

A careful analysis of Java programs shows an interesting pattern that is helpful in this case. According to [5], within an application, a small subset of instantiated objects will last to the end of the application, while all others have a short lifetime. This pattern enforces the idea that object references are usually localized, to a certain extent. With this information, it is possible to optimize the profiling process and achieve fast recognition of short living objects. In general, to automate the process we need some steps:

- 1 The user needs to identify all possible execution cases to be tested.
- 2 Execute the program profiler for the identified cases.
- 3 Analyze the profiler output, and generate the flow graph and objects reference graphs.
- 4 Use the information to place the *create* and *destroy* primitives at the proper places.
- 5 Use the ByteCode Engineering Library (BCEL) [2] to modify the classes and provide the proper constructors for the reusable objects.
- 6 Compile and run the modified program to test it before actual deployment.

Generally, the development process of real-time and embedded applications involves rigorous testing to ensure proper operation. Therefore, the profiling process we propose will not impose a high overhead on the developers. The framework provides a basic mechanism for automating the object-reuse technique that can be further explored and optimized to achieve the desired results.

6. Conclusions and Future Work

In this paper, we introduced a technique to eliminate garbage collection for real-time and embedded real-time Java, thus providing a deterministic execution behavior. The technique, object-reuse, provides a mechanism to reuse objects within the memory space to minimize the required memory size. With this technique the memory requirements can be limited to the actual size of memory available, thus eliminating the need to activate garbage collection. The object-reuse technique provides the user with simple primitives to create and destroy objects such that they can be used again in the application. It also limits the memory size needed, thus eliminating the need for activating GC if the assigned heap size is sufficient to hold the non-reusable objects and the reusable objects. In addition, it resulted in reducing the execution time of the applications due to the elimination of a large number of expensive object instantiations, which are replaced by the less costly object creation. Using the technique also provides more efficient utilization of the available memory, which is essential for embedded applications. In general, this enhancement provides a solution for one of the issues associated with having a real-time and embedded Java. Finally, we introduced the basic framework to automate this technique, thus removing the need for the developers to manually insert the primitives. In addition, the automated approach will also solve the problem of nested object references, which may not be done using the manual method. As a future phase, we intend to further investigate the suitable mechanisms to implement and optimize the automation of object-reuse. In addition, we would like to further investigate the impact of the automated technique

on the development, testing and operation of the applications using it.

Acknowledgement

We thank Dr. Steve Goddard for his support and valuable comments.

References

- [1] J. Al-Jaroodi and N. Mohamed, "Towards Efficient and Deterministic Memory Management for Real-Time Java", Technical report, TR03-02-01, University of Nebraska-Lincoln, 2003.
- [2] The Byte Code Engineering Library web page: <http://bcel.sourceforge.net/>
- [3] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, "The Real-Time Specification for Java", Addison-Wesley, Reading, Massachusetts, 2000.
- [4] L. Carnahan and M. Ruark, Editors, "Requirements For Real-time Extensions For the Java Platform, Report from the Requirements Group for Real-time Extensions For the Java Platform", NIST Special Publication, sep. 1999.
- [5] Tuning Garbage Collection with the 1.3.1 JavaTM Virtual Machine, Sun Microsystems, <http://java.sun.com/docs/hotspot/gc/>, 1999.
- [6] R. Jones and R. D. Lins, "Garbage Collection: Algorithms for Automatic Dynamic Memory Management" John Wiley and Sons. 1999.
- [7] N. Mohamed, A. Davis, X. Liu, and B. Ramamurthy, "JOR: A Java Object Router", in PDCS, pp 630-635, November 2002.
- [8] K. Nilsen, "Issues in the Design and Implementation of Real-Time Java", In Real-time Magazine, No. 1 1998, http://www.dedicated-systems.com/magazine/98q1/1998q1_p006.pdf
- [9] A. Petit-Bianco, "Determining suitable memory management algorithms", Dr. Dobb's Journal October 1998, <http://www.ddj.com/documents/s=915/ddj9810a/9810a.html>
- [10] Real-Time for Java Expert Group, web page: <http://www.rtg.org/>
- [11] M. Timmerman, "Java Here, Java There, Java Everywhere ...", In Real-time Magazine, No. 1, 1998, http://www.dedicated-systems.com/magazine/98q1/1998q1_p006.pdf
- [12] PERC real-time JVM from NewMonics, web page: <http://www.newmonics.com/perc/info.shtml>