Object-Reuse for More Predictable Real-Time Java Behavior

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

Abstract
One of the problems with Java for 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 Java programs. This technique can eliminate or reduce the need for GC and allows for a more predictable execution behavior and efficient utilization of the available memory. A theoretical model is presented and 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 in some cases GC can be eliminated and an application’s execution time decreases and becomes more predictable.

1. Introduction
For Java to support real-time applications, a number of issues must be taken into consideration [6]. One aspect is efficient and predictable dynamic memory management. Regular Java implementations use automatic garbage collection (GC) to clean-up unused objects. Moreover, real-time applications that generate a large number of objects can fill the available memory frequently. Therefore, GC is frequently executed to free it. This results in a non-predictable timing behavior of the application. One approach to limit the use of GC is to statically pre-allocate the memory needed by the application. However, this reduces memory utilization and restricts the number of tasks that can execute, which may reduce CPU utilization. Real-time applications such as application-level routers or on-line networked multimedia applications 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 remain in the heap until GC collects them, which causes unacceptable delays.

In this paper, we propose a technique to eliminate or reduce 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 other contexts to reduce memory usage for database and server applications; why not for real-time applications. By combining these two concepts, the non-predictable behavior of GC can be avoided. In this technique, the user is provided with basic primitives for creating and destroying objects and reusing the same heap space for new objects. Instead of creating a new object every time an object is needed, we use the existing unused objects that were previously instantiated from the same class. Thus, we partially transfer the job of GC to the user, while efficiently utilizing the memory using object reuse. In addition, this also reduces the execution time of the application by avoiding the relatively high cost of object instantiation.

2. Background and Related Work
Real-time applications have strict temporal requirements; therefore, it is desirable to reduce or eliminate non-predictability in the application and in the run-time environment supporting it. In addition, some real-time applications are limited in memory, thus require dynamic memory management, which introduces further non-predictability. Java has some non-predictable features such as GC; however, it is still considered for real-time applications. In 1999, The Requirements Group for Real-time Extensions for the Java Platform issued a report for RTJ requirements. However, a number of issues still need to be addressed [6]. These include predictable GC, faster execution, portability, dynamic adaptability and fault tolerance [14]. The Real-Time Java Expert Group produced specification (RTSJ) [15] for additions to enable real-time Java applications. TimeSys Inc. [17] built the official reference implementation of RTSJ and the technology compatibility kit (TCK). However, other implementations also exist such as Jbed RTOS Package [11], SimpleRTJ [16], PERC [14], and JamaicaVM [10].

In the non-real-time JVM, some developers relied on object pools and reuse [7] to reduce GC activities and enhance the performance of applications such as in WebSphere by IBM [5]. However, in real-time applications time constraints is the main issue. Thus, even a reduced rate of activating GC is not acceptable since it is not predictable. In RTJ, GC is a low priority thread; however, in reality, it halts the application to correctly execute. This causes a problem given the tight bounds on execution time. A study of the traditional GC algorithms can be found in [12]. GC for real-time systems usually falls into one of four categories; (1) Using efficient incremental and concurrent GC (PERC [9][14][18] and RTSJ [6]) and generational GC [8]; (2) Providing preemptable GC (as in RTSJ[6]); (3) Using static memory allocations; and (4) Providing options to disable GC for hard real-time tasks.

In addition, many techniques to tune and enhance GC performance such as profiling [5], sampling of object usage [1], using more efficient hashing techniques [3] and enhancing the precision of GC [2] have been proposed. On the other hand, some JVM implementations provide a selection of GC algorithms for the applications as in Sun’s JVM. Most
techniques rely on the existence of the GC in some form. The solution we propose combines some of the advantages of these techniques to eliminate GC and, at the same time, does not limit the memory utilization. However, our goal here is not enhancing the application performance; rather, we aim towards providing a predictable time bound on the application.

3. The Object-Reuse Technique

Here, we discuss an approach for efficient and predictable memory management for RTJ. The solution is based on providing mechanisms to reuse existing unneeded objects instead of instantiating new ones. The prototype implementation provides the developer with two primitives to create and destroy objects. A theoretical model of memory utilization using this technique is given.

The Solution Implementation: Many real-time applications repeatedly create multiple instances of an object to be used temporarily; however, these objects remain in the heap until GC cleans it. This will repeatedly invoke GC 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. Using these primitives, in place of object instantiation methods, leads to some desirable effects. (1) Eliminating (or in the worst case minimizing) the need for executing GC, thus removing the non-predictable factor in tasks’ execution time; (2) Providing efficient use of memory through reuse, this is important for real-time systems with limited memory and for systems managing many temporary objects; and (3) Reducing the cost of instantiating objects, since the cost of creating and destroying objects for reuse is significantly smaller than that of object instantiation. 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 activeObject (maintains references for in-use objects) and deletedObject (maintains references for destroyed objects) are maintained. When the create primitive is executed, it will first check if the object to be created is in deletedObject. If it is, the object reference is moved to activeObject and the object is re-initialized and returned to the user. Otherwise, a new object is instantiated 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 needs to modify the object constructors to allow for the re-initialization of reused objects.

The Model: To study the feasibility and effectiveness of our proposed solution, a memory allocation model is presented for applications that require repetitive use of a given set of objects during execution. An application’s memory needs are defined using the following attributes.

- \( M_0 \) is the total amount of memory needed continuously throughout the application lifetime.
- \( N_t \) is the number of objects used in \( M_0 \).
- \( P_t \) is the amount of memory that can be reused.
- \( N_t \) is the number of objects used in \( M_0 \).
- \( F_t \) is the frequency of creation of objects in the reusable memory. For example, if a loop instantiates \( N_t \) objects then the frequency \( F_t \) is the number of iterations in the loop.

Execution time and costs of instantiating, creating, or destroying objects are defined as follows.

- \( e \) is the total execution time of the application.
- \( C_t \) is the cost of creating a new object. This is generally high since the JVM needs to check the heap for available space to add the object and execute its constructor.
- \( C_d \) is the cost of creating an object. In our solution, creating an object requires changing the reference of an existing object (previously made available by instantiating it or by applying destroy) to be used for the newly created object.
- \( C_d \) is the cost of destroying an object including changing the reference an unused object.

Let \( M_0 \) be the total heap size allocated for the application and \( M_t \) be the total memory demand of the application. \( M_t \) can be calculated as the sum of the continuously used memory (\( M_u \)) and the amount of memory that is repeatedly used (\( M_r \)) multiplied by the frequency of using that memory (\( F_t \)) as \( M_t = M_u + (F_t \times M_r) \). This is based on regular JVM behavior where objects are not reused. Using this information, we can state the following theorem:

**Theorem 1:** Given the total heap size assigned for an application \( M_0 \) and the total memory demand of the application \( M_t \), GC is guaranteed not to execute if and only if \( M_t \leq M_0 \).

**Proof:** The proof is based on understanding the mechanisms of objects use and the conditions for initiating GC. GC is executed if and only if one of the following conditions occurs: There is not enough space for a new object; The constant string table is full; The references table is full; There is not enough space for a new method frame; or The user invokes GC using the System.gc() static method. The components are all stored within the application heap. When an application starts executing, at any given point in time let the used part of the heap be \( M_d \). If the application instantiates a new object of size \( M_s \), the system will check the available space in the heap. If \( M_d + M_s \leq M_0 \), then the object is created, added to the heap, and used heap size \( M_d \) is adjusted to \( M_d = M_d + M_s \) and if \( M_d + M_s > M_0 \), then GC is initiated to free some space. Similarly, if a new method in an object is invoked and requires \( M_m \) space for the method frame, the system will check for the necessary space in the heap. If \( M_d + M_m \leq M_0 \), then the object is created, added to the heap, and used heap size \( M_d \) is adjusted to \( M_d = M_d + M_m \) and if \( M_d + M_m > M_0 \), then GC is initiated to free some space. Furthermore, access to the constant string table and the reference tables will only invoke GC if they cause memory utilization to exceed the heap size. In standard JVM, these are the only conditions required to initiate GC. Therefore, GC is only used if additional space for the objects, methods or their references is needed. However, if the total amount of memory needed for the application is known \( M_0 \), then the condition \( M_t \leq M_0 \) is necessary and sufficient to determine if GC will be initiated during the execution time.

Based on Theorem 1, it is possible to eliminate the need for GC by initially allocating a heap of size \( M_0 \) for the application.
However, this is not a practical solution for three reasons. First, the frequency of creating reusable or temporary objects $F$, may not be known before hand, thus $M_r$ cannot be accurately calculated or will be over estimated. Second, if $M_r$ was determined before hand, it will usually be vary large; thus, it is not practical to allocate a huge portion of memory to a single application. Finally, if $M_r$ is too large, the system may not be able to provide the required amount of memory, thus there will be no guarantees that the application will not fail. In the solution proposed, we allow for object reuse, thus reducing the total amount of memory $M_r$ needed for the application. In this case, we can calculate $M_r = M_0 + M_1$, thus eliminating the multiple instantiations of new objects. According to Theorem 1, the GC will be eliminated if and only if the system can allocate memory of size $M_r$ for the application. However, even if the system cannot afford to allocate the total amount of memory, the GC operation will be reduced by a fraction of $\frac{M_u + M_r}{M_u + M_r + F}$, due to the smaller total memory needed.

Another issue to address is the costs of instantiating new objects and the cost of creating and destroying objects in the proposed solution. Let $C$ be the cost of instantiating all objects used in the application. For the regular JVM

$$C = N_r C_r + C_i \left[ N_r + N_r F_r - N_r \right]$$

However, with the object-reuse technique, the cost is

$$C = N_r C_r + C_i \left[ N_r + (F_r N_r - N_r) \left( C_e + C_i \right) C_r \right]$$

By comparing $\frac{(F_r N_r - N_r) (C_e + C_i)}{C_r}$ in the second equation above to $N_r F_r - N_r$, in the first one, the first term is smaller since $C_r + C_i$ is very small compared to $C_r$ This is because the cost of instantiating a new object is higher than the total cost of creating and destroying an object, thus the total cost incurred using the reuse method is smaller than that in the regular method. In general, the proposed solution limits the required memory for the application and helps reduce the total cost of using objects in the application. However, the effect becomes more pronounced in applications that, by nature, need to repeatedly use objects for intermediate (temporary) processing during their execution lifetime. For the application developer, determining the number of objects needed in this manner (long term and reusable objects) is easier than trying to determine the entire number of objects to be used. With this information, the total memory needed can be assigned statically before execution and GC will be guaranteed not to execute.

4. Evaluation

To measure the performance of the object-reuse technique, a number of benchmark examples are used on Java SDK 1.4 and PERC (real-time JVM). The first set of experiments used a micro-benchmark to measure individual components such as response time of tasks with and without object-reuse. In addition, an application benchmark, Java Object Router (JOR) [13] is used to show the overall performance.

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. Heap size was fixed to 8MB and the image object is binary of size 2000x2000 bits requiring 500,000 bytes of memory. Two versions of GC (full and incremental) using SUN JVM were measured. Using the full GC, the object-reuse version had 0 activities compared to 51 without object-reuse, thus resulting in shorter overall execution time for the object-reuse version (5.1s vs. 6.2s). Similar results were obtained when incremental GC was used; however, the GC activities were much higher (258) compared to the full GC. As a result, the object-reuse version executed faster too (5.1s vs. 6.7s). 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 was used with active GC, the time for the regular program (8.8s) was higher than the object-reuse program (7.6s). With GC disabled, the regular program failed due to insufficient memory, but the object-reuse version continued normally (taking 7.7s). This shows that the reuse technique allows the program to continue executing within the limited memory available.

The second experiment measures the execution time of the iterations in the programs one at a time. A description and full results of these experiments can be found in [4]. 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 [4]. 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.

Application Benchmark: JOR [13] is an application level router that routes Java objects among distributed Java applications on a network based on object type, contents, or 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 predictable routing process delay under certain workloads. We used time stamps for each object as soon as it received and when it is ready to be transferred, where the difference between the two times is the processing time of the object. This time 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 1 show that 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 attributed to GC activities that are triggered non-predictably. On the other hand, the execution time with the object-reuse technique has much less variation (2 to 6 ms). Moreover, the average execution time of the regular Java program is 5.78 ms per packet and it drops to 2.41 ms 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 technique along with the elimination of GC activities.

**Discussion:** 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 is completed. In this case, object-reuse becomes very useful. Moreover, the loop in the benchmark represents the service rate of the application, but we only included one object in the loop to clarify the measurements. In real applications, there may be multiple objects created at different rates and the repetition may be in the form of loops or multiple threads. Another experiment was conducted to measure the costs of instantiating a new object and creating and destroying an object in the object-reuse technique. The execution times observed show object reuse and destruction always cost less than 0.001 second, while the object instantiation takes 0.02, 0.15, and 0.015 seconds on JDK 1.4, PERC /w GC, and PERC /wo GC, respectively. Applying these figures in the model to calculate C, we get the following results; (1) Using SUN JVM: ImageGC takes 1.02 seconds and ImageOR takes 0.138 seconds, and (2) Using PERC JVM: ImageGC takes 0.765 second and ImageOR takes 0.128 seconds. In both cases, the total cost of using the objects is much smaller with the object-reuse technique. This is due to the savings gained by not instantiating new objects in every iteration. To summarize, the measurements obtained in these experiments show that the object-reuse technique did not only reduce (or eliminate) the need for GC, but it also reduced the overall execution time of the applications. The results are encouraging and motivate further work to enhance and automate the technique.

**Table 1** The results of JOR experiment

<table>
<thead>
<tr>
<th>Router with GC</th>
<th>with OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Packets processed</td>
<td>100</td>
</tr>
<tr>
<td>Average time per packet</td>
<td>5.78</td>
</tr>
<tr>
<td>Min. packet processing time</td>
<td>3</td>
</tr>
<tr>
<td>Max. packet processing time</td>
<td>30</td>
</tr>
<tr>
<td>Difference (Max – Min)</td>
<td>27</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.07</td>
</tr>
</tbody>
</table>

5. Conclusions and Future Work

In this paper, we introduced a technique to eliminate garbage collection for real-time Java applications, thus providing a predictable execution behavior. The technique, object-reuse, provides a mechanism to reuse objects within the heap to minimize the required memory size. Our technique provides the users with simple primitives to create and destroy objects. It also limits the memory size needed, thus eliminating the need to activate GC if the assigned heap size is sufficient to hold all the non-reusable and reusable objects. In addition, it resulted in reducing the execution time of the applications due to eliminating 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 memory, which is essential for most real-time applications. The theoretical model and experiments show the characteristics and performance of the object-reuse technique. In general, this enhancement provides a solution for one of the issues associated with having a real-time Java. The next step is to introduce a framework to automate the technique, thus relieving developers from manually handling the primitives. Another task to accomplish is to further investigate mechanisms to implement and optimize the automation of object-reuse.

**Acknowledgement**

We thank Dr. Steve Goddard for his support and valuable comments. Also our gratitude to Dr. Chris Gill, for his valuable suggestions during the preparation of the final manuscript.

**References**


