Memory Management Design to the Concurrent Execution of RTSJ Applications*

M. Teresa Higuera-Toledano

Faculty of Computer Science, Complutense University of Madrid,
Ciudad Universitaria, 28040 Madrid Spain
mthiguer@dacya.ucm.es

Abstract. Our objective is to adapt the Java garbage collection to an embedded system, e. g., wireless PDA executing multimedia applications. We consider the concurrent execution of several applications within a single JVM, giving an approach to divide/share the memory among the applications executing concurrently in the system. Since a multimedia application is supported by several tasks, some of them being response time limited, while others being high performance oriented, we must consider the real-time characteristics of the GC. In order to have a different GC per application, we introduce and define application-specific memory, building upon the Real-Time Specification for Java (RTSJ) from the Real-Time Java Expert Group.

1 Introduction

Demands for multimedia services in embedded real-time systems, such as wireless Personal Digital Assistants (PDAs), are increasing. The use of PDAs is foreseen to outrun the one of PCs in the near future. However, for this actually happen there is still the need to devise adequate software and hardware platforms in order to not overly restrict the applications that are supported. In general, the environment must accommodate the embedded small-scale constraints associated with PDAs, and enable the execution of the applications traditionally supported on the desktop such as soft real-time multimedia applications that are becoming increasingly popular. In particular, it is mandatory to finely tune the management of memory consumption, and to enable provisioning new applications extending the capabilities of the mobile phones.

Addressing the above requirements then lies to devising resource management policies taking into account both real-time constraints and the available memory budget, and offering an open software environment which enables extending the application set that can be supported by the PDA. The ideal candidate to providing an open environment is Java, which appears as a major player in the area of embedded software environment, and allows us to get portable code, which can possibly be dynamically downloaded. Although, Java has some shortcomings regarding the target device, that have been solved by extending Java API to meet the requirements

* Founded by the Ministerio de Ciencia y Tecnología of Spain (CICYT); Grant Number TIC2002-00334.

R. Meersman and Z. Tari (Eds.): OTM Workshops 2003, LNCS 2889, pp. 479–489, 2003.

appertained to embedded real-time software [4] and [11], such as real-time scheduling and predictable memory.

Therefore the need for executing multiple applications in the same JVM is increasing [2], [5]. When executing concurrently multiple applications, if an application consumes all the available memory, the other applications get starved. One way to avoid this problem is to divide the memory among running applications in the system, giving to each application a separate garbage collectable area. Hence, the partition of the heap in separate sub-heaps allows: invoking several collectors concurrently; having a collector per sub-heap that is customized according to the behaviour of the embedding application, minimizing the latency time to preempt a local collector from the CPU when a high priority task from another application arrives, and distributing the collector overhead among activities.

From a real-time perspective, the Garbage Collector (GC) introduces unpredictable pauses that are not tolerated by real-time tasks. Real-time collectors eliminate this problem but introduce a high overhead. An intermediate approach is to use Memory Regions (MRs) within which both allocation and de-allocation are customized and also the space locality is improved. Application of these two implicit strategies has been studied in the context of Java, which are combined in the *Real-time Specification for Java* (RTSJ) [4]. The J Consortium solution [11] proposes allocation contexts allowing us to group core objects that are free upon disposal of the allocation context. The core objects are not relocated and garbage collected, and core methods do no include code for synchronization with the GC.

This paper focuses on a memory management solution in order to divide/share the heap among different real-time applications accounting for the RTSJ.

1.1 The RTSJ Memory Model

The `MemoryArea` abstract class supports the region paradigm in the RTSJ specification through the three following kinds of regions: *(i)* immortal memory, supported by the `ImmortalMemory` and the `ImmortalPhysicalMemory` classes, that contains objects whose life ends only when the JVM terminates; *(ii)* (nested) scoped memory, supported by the `ScopedMemory` abstract class, that enables grouping objects having well-defined lifetimes; and *(iii)* the conventional heap, supported by the `HeapMemory` class. Objects allocated within immortal regions live until the end of the application and are never subject to garbage collection. Objects with limited lifetime can be allocated into a scoped region or the heap. Garbage collection within the application heap relies on the (real-time) collector of the JVM.

RTSJ also makes distinction between three main kinds of tasks: *(i) low-priority* that are tolerant with the GC, *(ii) high-priority* that cannot tolerate unbounded preemption latencies, and *(iii) critical* that cannot tolerate preemption latencies. Low-priority tasks, or threads, are instances of the `Thread` class, high-priority tasks are instances of the `RealtimeThread` class, and critical tasks are instances of the `NoHeapRealtimeThread` class. Synchronization among critical tasks and non-critical ones is problematic, since the non-critical may cause delays to the critical ones due to the execution of the GC. RTSJ solves the synchronization problem by introducing unsynchronized and non-blocking communication, e.g., the

`WaitFreeReadQueue` (resp. `WaitFreeWriteQueue`) class supports a *read-free* (resp. *write-free*) queue, which is unidirectional from non-real-time to real-time.

1.2 Paper Organization

The rest of this paper is organized as follows. Section 2 presents the general guidelines of our solution design to execute concurrently several applications. Section 3 details the characteristics of the local GC embedded in each application. Section 4 discusses resource management policies taking into account both real-time constraints and the available memory budget. Section 5 sketches an overview of related work. Finally, Section 6 offers some conclusions.

2 Extending RTSJ to Support Several Applications

In this section, the memory management model of RTSJ is extended to offer a multi-process execution. In the proposed solution, some memory regions are accessible by all the activities in the system, which allows inter-process communication by using both the communication model of Java based on shared variables and monitors, and the classes that the RTSJ specification provides to communicate real-time tasks and non-real-time threads.

2.1 Extending the MemoryArea Hierarchy

In order to obtain multi-process execution, we introduce the `MemorySpace` abstract class supporting three subclasses: the `CommonHeap` and `CommonImmortal` to support public memory without application access protection, and `ProtectedMemory` to define application-specific memory with access protection. There is only one object instance of both the `CommonHeap` and the `CommonImmortal` classes which are created at initialization system time and are resources shared among all activities in the system. In contrast, a new `ProtectedMemory` object is created when creating a new application and is a local resource protected from accesses of all the other activities in the system.

Creating a protected memory space implies the creation of both the local heap and the local immortal memory regions of the corresponding application. As in RTSJ, an application can allocate memory within its local heap, its immortal region, several immortal physical regions, several scoped regions, and also within the common heap and the common immortal spaces.

In order to obtain the reference of the common heap and the common immortal region in a similar way to the RTSJ model, we introduce the `instance()` method in both the `CommomHeap` and `ImmortalMemory` classes.

2.2 Sharing Memory

To facilitate code sharing, classes are stored within the common immortal space (i.e., the `CommonImmortal` object). In this way, all applications in the system access both code and data (i.e., *class variables*), of all classes. But there is a problem with the access to the class variables, declared as `static` in Java. These variables must be shared by all the tasks of an application, but they must be protected from the access of other activities. Thus, we maintain a copy of the class variables in the local immortal memory of the application. As in the solution given in [5], we maintain a copy of the class variables for each application using the class, while maintaining only a single version of the class code.

The same problem arises with *class monitors* (i.e., shared code related to synchronization), these methods are declared in Java as `static synchronized`. When a task enters a class monitor and is suspended by another task, if both tasks are from the same application, there is no problem. The problem arises if the two tasks are from different activities. To ensure mutual exclusion among tasks from the same application, while avoiding other activities to be affected, each application must maintain a separate copy of the monitor. The solution is then to allocate also in the immortal memory of the application a copy of the `static` code. This solution requires modifying the class loader to allocate in the immortal memory of each application, a copy of data and code declared with the `static` statement [5]. In general, there is a problem with resources shared among all tasks of an application that must be isolated from tasks of the other application.

2.3 Dealing with Critical Tasks

Whereas high-priority tasks require a real-time GC, critical tasks must not be affected by the GC, and as a consequence cannot access any object within the heap [4]. A reference of a critical task to an object allocated in the heap causes the `MemoryAccessError()` exception, which can be achieved by using *read barriers*. Note that read barriers occur upon all object accesses, which means upon executing both types of bytecodes: those causing a *load reference*[1] and those causing a *store reference*[2].

Note that read barriers are strictly necessary only when using read barrier-based collectors (i.e., incremental copying without handlers). When using a non-read barrier-based collector (i.e., mark-and-sweep, generational, copying-based using handles, and reference-counter), load operations do not affect the GC data structure. As consequence, the GC does not causes delays to critical tasks when accessing objects. Hence, we apply the same optimization as for the incremental GC which is to use write barriers instead of read barriers [2]. Since reads do not interfere with the GC, the restriction on critical tasks can be reduced to write barrier checks. And, the `MemoryAccessError()` exception, which raises when a critical task attempts to

---

[1] `getfield`, `getstatic`, `agetfield_quick`, `agetstatic_quick`, or `aaload` bytecodes.

[2] `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, or `aastore_quick` bytecodes.

access an object within the heap, is changed by the `IllegalAssignmentError()` exception, which raises when a critical task attempts to assign an object that belongs to the heap [7].

3 Garbage Collection in an Embedded Multimedia Framework

Different GC techniques are appropriate depending on the real-time embedded application. If the application does not generate cyclic data structures, a reference-counting GC algorithm is the most appropriate. Alternatively, a generational GC collects the younger objects more frequently than the older ones. While real-time GCs provide the worst-case guarantees, generational GCs improve the average performance at the expense of the worst-case. Generational collectors may be good for some applications, which are soft real-time, like multimedia ones.

3.1 The Basic Collector Strategy

The characterization of the applications behaviour related with dynamic memory allocation helps choosing the local GC technique. There are also some important considerations when choosing a real-time GC strategy, among them are space cost and barrier costs. Copying GCs require doubling the memory space, because all the objects must be copied during GC execution. Non-copying GCs do not require this extra space, but are subject to fragmentation. We specifically consider the incremental non-copying GC based on the tri-color *treadmill* algorithm [1] that has been described in [7]. This algorithm allows the interleaved execution of the collector and the application[3], which execution is synchronized by using write barriers to detect whether the application updates pointers. When the collection is completed, objects that must execute the `finalize()` method are moved to the finalize-list, which are executed by a specialized thread such as in [15]. Finally, for objects that have finalized, their memory is freed. Then, a compacting phase can be added to move objects into a continuous block into the heap, which implies some degradation of real-time guarantees.

Alternatively, we can use a generational copying collector such the employed by the Java *HotSpot* VM, which uses a GC based on the *train* algorithm [9] to collect the old space, which divides the old object space into a number of fixed blocks called *cars*, and arranges the cars into disjoint sets (*trains*). This technique uses write-barriers to trap both the inter-generation references (pointers form objects within the old generation to objects within the *new* generation), and references across cars within the same train.

We can improve the performance of the above collectors using the hardware write barrier support that the picoJava-II microprocessor [17] provides. From the standpoint of GC, this microprocessor checks for the occurrence of two type of write barriers: reference-based used to implement incremental collectors, and page-based designed specifically to assist generational collectors based on the *train-based* algorithm.

---

[3] Each small GC unit interleaved with the application execution is called *increment*.

3.2 The GC and MR

Since objects allocated within regions may contain references to objects within the local heap, a tracing-based GC (e.g., incremental [1] or generational [9]) must take into account these external references, adding them to its reachability graph. When an object outside the heap references an object within the heap, we must to add the object that make the reference to the root-set of the collector, which is achieved by using write barriers. When the collector explores an object outside the heap (i.e., a root), which has lost its references into the heap, it is eliminated from the root-set. Finally, when a scoped MR ends, all objects within the region having references to the objects within the heap are removed from the root-list of the collector, and all the objects within the region are moved to the finalize-list.

3.3 Running the Local GC

In RTSJ, an instance of the abstract `Scheduler` class implements a scheduling algorithm, providing flexibility to install an arbitrary scheduler. The `PriorityScheduler` subclass contains the base scheduling, which is preemptive and priority-based, having at least 28 real-time priority levels plus the 10 traditional Java threads priorities. Since fixed-priority scheduling with preemption is the most commonly used in real-time system construction, we adapt our real-time collector to this scheme by establishing the following main priority levels:

1. Low-priority task are mapped in the 10 Java priority levels.
2. The incremental GC is mapped in the lowest real-time priority level (i.e., the 11 priority level).
3. High-priority tasks are mapped in the lowest real-time priority levels, and can be interleaved with the GC (e.g., the 12-38 priority levels).
4. Critical tasks are mapped in the highest real-time priority levels, and can preempt the GC at any instant (e.g., the 12-38 priority levels).

When creating a task, in addition to the memory parameters, the memory region, and the runner code, we can specify both a `PriorityParameters` object and a `ReleaseParameters` object to control the task behavior. Figure 1 shows how we can create a task executing the incremental GC algorithm. Where the increment value is computed as the worst case execution time of a GC *increment* and is used to specify both parameters its *worst case execution time* and its *deadline*.

```
RealtimeThread gc = new RealtimeThread(
    new PriorityParameters(PriorityScheduler.getMinPriority()+10),
    new AperiodicParameters(increment, increment, null, null),
    null, null, null,  new MyIncrementalGC());
```

Fig. 1. Scheduling the GC as a task in RTSJ.

4 Resource Management and Memory Negotiation

Resource management allows making real-time systems safe and extensible, and can be studied from three points of view: the ability to allocate resources to an application (*resource allocation*), the ability to track resource usage (*resource accounting*), and the ability to reclaim the resources of an application when it terminates (*resource reclamation*). There are two costs associated with the heap allocation, the direct cost of the `new()` method and the cost to perform the GC. In programs with high allocation rate, like multimedia applications, this cost can be substantial. Resource accounting is influenced by the way in which activities obtain services, and is difficult when there are shared system services. And resource reclamation presents some problems when a task terminates exceptionally.

4.1 Resource Allocation

The partition of memory allows us to invoke several collectors concurrently, where the reclamation rate can be different for each application. When an application allocates objects, the assigned memory is not necessarily continuous, e. i., partitions are not physical[4]. The execution of applications further relies on a negotiation protocol as commonly used when running multimedia applications. The handling of real-time constraints together with the limited capability of the PDA requires making sure that there is enough resource available to execute a new application. In the case where there is not enough resource left, it is common to have a negotiation protocol taking place between the application and the system where the application lowers its resource requirements by changing the resulting quality of service offered to the user. We are interested in addressing dynamic negotiation in order to benefit from resources left by applications that terminate.

Considering the RTSJ solution, prior to starting a task, a `MemoryParameters` object must be assigned to it. The memory requirements of a task are used for both the executive (to control admission) and the GC (to satisfy all tasks allocation rates). When a task exceeds its memory resource limitation, the executive can generate an exception to reject workloads. This solution allows communicating the memory requirements of a task to the system, but we can not change these requirements during the live time of the task. We found interesting that applications can negotiate dynamically with the real-time executive for resources. The resources budgets that we consider are: memory size and memory allocation rate. Upon arrival a new application, it must reserve resources. If there are not enough resources available, the application must revise its requirements. If still there are not resources available, the memory requirements of all applications can be revised. Finally, the incoming application will be either, accepted or rejected according to the results of the negotiation.

[4] As an exception, the memory assigned to `LTMemory` objects must be continuous, because this kind of regions provides linear allocation time.

4.2 Resource Shared and Communication

Resource management is related with the communication model. The standard JVM supports a *direct sharing* communication model, which makes difficult resource reclamation (e.g., when an application ends and has shared objects with tasks of other applications). To communicate two tasks of different activities, both critical or both non-critical, we present a *limited sharing* model, where shared objects must be allocated within the common memory. Hence, this model is less flexible than direct sharing. However when an application ends, all objects are reclaimed without problems. For object allocated within the common spaces, class variables and class monitors are allocated within the common immortal memory. When applications have no address space shared, *copying* is the only possibility. This solution is the most flexible and the most adequate for a distributed real-time Java extension.

Communication among tasks which belong either to the same or different activities is safe, when both tasks are critical or when both tasks are non-critical. But communication when a task is critical and the other one non-critical require unsynchronized and non-blocking operations which make it unsafe. Note that when both tasks are from different activities, there is no problem with the local collector. However, the problem still exists with the common collector. Then, communication among critical and non-critical tasks of different applications must be achieved by using the common heap as the *memory area* parameter in the constructor of the *wait-free queues*.

4.3 Resource Reclamation

Each application heap is collected by a local collector (e.g., the GC described in Section 3), while the objects shared by the applications are collected by an incremental collector based on the reference-counting technique. Then, some form of distributed information to collect shared objects is required. We maintain a list of pointers, called *external-references*, to objects within the common heap having references from objects within other memory spaces. And for each object within the common heap, we maintain a counter giving the number of links to the object. An attempt to create a reference to an object into a field of another object requires three different treatments depending on the space to which the object belongs:

- Treatment A: the reference is within the common heap. It is not allowed for critical tasks, therefore the write must be aborted and the `IllegalAssignemt()` must be triggered. For non-critical tasks, the write must take actions for the reference-counter collector, and also it causes the creation of an external-reference whether the X object is allocated outside the common heap (i.e., for inter-space references).
- Treatment B: the reference is within the common immortal memory. It is allowed and nothing needs to happen.
- Treatment C: the reference is within a protected. We distinguish two cases. For intra-spaces references (i.e., space(X)=space(Y)), we must take into account the assignment rules imposed by RTSJ [3] and actions needed by the local GC whether the reference is within the protected space [8]. For inter-spaces

references, the write must be aborted and the `IllegalAssignemt()` must be triggered for inter-region references.

5 Related Work

The literature already provides us with a number of base solutions upon which we can build to meet our objectives. Focusing on work from the operating system community, relevant work on the issue of resource reservation has been examined in [14]. The proposed solution lies in a real-time JVM that interfaces with the abstractions of a resource kernel to build a resource-based JVM semantically compliant with RTSJ. Compared to this work, we are interested in examining management of resource consumption taking into account both real-time constraints and the available memory budget. Unlike traditional JVMs, in our proposed solution, the applications run concurrently within a single JVM instance, in a way similar to the Java Os from Utah [2]. Like in this solution, in order to provide secure and controlled accesses, we limit direct sharing among activities. When two activities want to communicate, they must share an object residing in the common heap. We take this solution as a trade-off between a more general solution such as allow activities to communicate using the RMI, and forbidding all possible communication.

The main applications for the PDAs are typically those run during a trip. In this context, applications will be Internet-based for accessing both discrete and continuous multimedia data. Internet services contain active code rather than static data, which raises serious trust and security issues. In order to provide strong isolation between services, both to enforce security and to control resource consumption. In [18], a new kernel architecture to isolate un-trusted applications has been proposed, whose function is to subdivide a physical machine into a set of fully isolated protection domains, and where each virtual machine is confined to a private namespace. Since Internet services are designed and operated by independent users, sharing is considered infrequent, and isolation has been strengthened.

A reconfigurable virtual machine supporting multiple user environments with varying degrees of criticality and privilege has been presented in [12]. This architecture provides hardware-enforced guarantees of resource separation, and it is based on the JEM-1 Java-based microprocessor. Hardware-based Java platforms (e, g., [6]) provide efficient support for bytecode execution, hard real-time, and also safe and secure multiple virtual machine execution.

In [10], we found a study of the requirements for embedded software environment aimed at wireless PDAs. These requirements have been addressed through a middleware platform, where the main issues in offering services for resource management lie in customized memory management. A solution to resource management and negotiation has not yet been addressed by the NIST group. However, its recommendation is to negotiate resources via the API. The PERC executive [13] analyzes availability of resources and determines when to accept a new real-time activity through a representation of the activity's *minimum* and *desired* resources, which allows resource negotiation.

6 Conclusions

This paper has presented a memory management design solution for extending the RTSJ specification to execute several activities concurrently in the same JVM. Two issues that we have addressed are how the memory is shared/divided among applications, and how allocation and deallocation are managed. To facilitate code sharing, classes are stored in the immortal common space. In order to provide secure and controlled access to the common memory regions and to maintain the assignment rules of RTSJ, we use a write barrier strategy. The logical distribution of the memory among applications can be based on a negotiation protocol, as typically used by multimedia systems. Regarding the customization of the collector running within each application heap, its selection depends on the features of the hosted application, and is left upon the responsibility of the developer (possibly aided by adequate analysis tools). For example, if the application does not generate any cyclic data structure, a reference counting GC algorithm is the most appropriate.

Our proposal builds upon existing work since the area of memory management. The contribution of our work comes from the adaptation and integration of relevant solutions in the context of RTSJ. This work need to be extended in multiple directions. Firstly, the proposed solution needs to be implemented and experimented with. Therefore, a memory allocation profile and techniques to determine the memory usage profile are very important issue requiring study.

References

[1] H.G. Baker. “The Treadmill: Real-Time Garbage Collection without Motion Sickness”. In Proc. of the Workshop on Garbage Collection in Object-Oriented Systems. OOPSLA'91. ACM 1991. Also appears as SIGPLAN Notices Vol. 27, no. 3, 1992.

[2] G. Back, P. Tullmann, L. Stoller, W.C. Hsieh, and J. Lepreau. Java Operating Systems: Design an Implementation.. Technical report, Department of Computer Science, University of Utah, http://www.cs.utah.edu/projects/flux, August 1998.

[3] W.S. Beebe and M. Rinard. “An Implementation of Scoped Memory for Real-Time Java”. In Proc of 1st International Workshop of Embedded Software (EMSOFT), 2001.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. (*The Real-Time for Java Expert Group*). "Real-Time Specification for Java". RTJEG 2002. http://www.rtj.org

[5] G. Czajkowski. Application Isolation in the Java Virtual Machine. In Proc. of Conference on Object and Oriented Programming, Systems Languages and Applications, pages 354–366. OOPSLA, ACM SIGPLAN, October 2000.

[6] Hardin D.S. “Real-Time Objects on the Bare Metal”. An Efficient Hardware Realization of the Java Virtual Machine. Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2001.

[7] M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Memory Management for Real-time Java: an Efficient Solution using Hardware Support". Real-Time Systems journal. Kluber Academic Publishers, to be published.

[8] M.T. Higuera and M.A. de Miguel. “Dynamic Detection of Access Errors and Illegal References in RTSJ”. In Proc. Of the 8th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS). IEEE 2002.

[9] R. Hudson and J.E.B. Moss. Incremental Garbage Collection of Mature Objects. In Proceedings of the first International Workshop on Memory Management. September 1992.

[10] V. Issarny, M. Banatre, F. Weis, G. Cabillic, P. Couderc, M.T. Higuera, and F. Parain. Providing an Embedded Software Environment for Wireless PDAs. In Proc. of the Ninth ACM SIGOPS European Workshop – Beyond the PC: New Challenges for the Operating System, September 2000.

[11] J. Consortium Inc. "Core Real-Time Extensions for the Java Platform". Technical Report. New-Monics Inc. http://www.j_consoutium.org. 2000.

[12] D.W. Jensen, D.A. Greve, and M.M. Wilding. Secure Reconfigurable Computing. Advanced Technology Center Advanced Technology Center. http://www.klabs.org/richcontent/MAPLDCon99

[13] K. Nilsen. Adding Real-time Capabilities to Java. Communications of the ACM, 41(6), June 1998. pag. 49–56.

[14] D. de Niz, R. Rajkumar. Chocolate: A Reservation-Based Real-Time Java Environment on Windows/NT. In Proc of Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000). June 2000.

[15] Petit_Bianco and T. Tromey. Garbage Collection for Java in Embedded Systems. Proceedings of IEEE Workshop on Programming Languages for Real-Time Industrial Aplications. December 1998.

[16] Sun Microsystems. "The Java HotSpot Virtual Machine". Technical White Paper. 2001. http://java.sun.com.

[17] Sun Microsystems. "picoJava-II Programmer's Reference Manual". Technical Report. Java Community Process, May 2000. http://java.sun.com.

[18] Whitaker, M. Shaw, and S.D. Gribble. Denali: A Scalable Isolation Kernel. In Proceedings of the Tenth ACM SIGOPS European Workshop, Saint-Emilion, France, September 2002.

[19] Wilson P.R. and Johnston M.S. "Real-Time Non-Copying Garbage Collection". ACM OOPSLA Workshop on Garbage Collection and Memory Management. September 1993.