Illegal References in a Real-Time Java Concurrent Environment*

M. Teresa Higuera-Toledano

Facultad Informática, Universidad Complutense de Madrid, 28040 Madrid Spain

Email: mthiguer@dacya.ucm.es

Abstract

We consider the concurrent execution of several applications within a single JVM. 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. The RTSJ memory model imposes strict assignment rules to or from memory areas preventing the creation of dangling pointers. An implementation solution to ensure the pointer safety of Java consists to check the imposed rules before executing each assignment statement by using write barriers.

1 Introduction

In this paper, we focus on a memory management solution in order to divide/share the heap among different real-time applications accounting for relevant Java specifications: the application isolation API [5] and the *Real-time Specification for Java* (RTSJ) [4], currently under revision as JSR-121 and JSR-001 respectively.

The application isolation API [5] guarantees strict isolation between programs (*isolates*). An isolate encapsulates an application or component, having its own version of a static state of the classes that it uses. Isolates have disjoint objects graphs, and sharing objects among two different isolates is forbidden. The `Isolate` class allows us to create an isolate by specifying a class.

In RTSJ, the `MemoryArea` abstract class supports the memory region paradigm 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. The lifetime of objects allocated in scoped regions is governed by the control flow. Strict assignment rules prevent the creation of dangling pointers, which violation causes the `IllegalAssignment()` exception.

The organization of the paper is as follows: we first present an approach to isolate the application memory, and to detect whether the application attempts to create an illegal assignment (Section 2). We propose a strategy to collect the local heap of each application, improving the performance of critical tasks (Section 3). We compare our research to related works (Section 4). Finally, some conclusions and a summary of our contributions; conclude this paper (Section 5).

2 Supporting 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 objects are accessible by all the applications 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 memory hierarchy

In order to obtain multi-process execution, we introduce the `MemorySpace` abstract class supporting two subclasses: the `CommonMemory` class 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 the `CommonMemory` class which is created at initialization system time and is a resource shared among all applications 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 applications 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. An application can allocate memory within its local heap, its immortal region, several immortal physical regions, several scoped regions, and also within the common memory space.

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

2.2 Sharing memory

To facilitate code sharing, classes are stored within the common immortal space (i.e., the `CommonImmortal.instance()` 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.

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 applications. 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 [10]. Then, we maintain a copy of both class variables and class monitors in the immortal region of each application using the class, while maintaining only a single version of the class code. This solution requires modifying the class loader.

2.3 Communication among applications

In order to share objects among applications, we introduce the `SharedObject` class and the `Shareable` interface that are equivalent to the `RemoteObject` class and `Remote` interface of the Java RMI. Objects allocated within the common memory are restricted to be shared objects (i.e., common objects belong to the `SharedObject` class or any of subclasses). Non shared objects can only be referenced locally. We can compare a shareable method to the synchronize methods of Java. Shared objects can be accessed only by invoking the methods declared in a shareable interface. This restriction avoids that two tasks access the same object concurrently without synchronization, which makes programs robust. Since shareable interfaces provide synchronization among tasks of either the same or different applications, to ensure mutual exclusion we allocate both shared objects and shareable interfaces within the common memory.

2.4 Introducing write barriers

An attempt to create a reference to an object into a field of another object requires a different treatment depending on the space to which the object belongs:

- Treatment A: the reference is within a protected space. We must take into account the assignment rules imposed by RTSJ [4] (i.e., objects within the local heap or the local immortal memory cannot reference objects within a scoped region, and objects within a scoped region cannot reference objects within another scoped region that is neither non-outer nor shared).
- Treatment B: the reference is within the common immortal memory. It is allowed and nothing needs to happen.

In Figure 1, we can see the write barrier pseudo-code that we have to introduce in the interpretation of the `putfield`, `aastore`, and `putstatic` bytecodes. We denote as X the object that makes the reference, and as Y the referenced object. The `spaceT()` function returns: `common` or `protected` depending on the type of the space to which the object parameter belongs. The `regionT()` function returns: `heap`, `immortal`, or `scoped` depending on the type of the region to which the object belongs. The `shared()` function returns `true` when the region of the object is a shared one. And the `nested(X,Y)` function returns `true`, when the region's scope of the Y object is the same or outer than the region of the X object [5].

```
if (spaceT(Y)=protected) and (space(X)<>space(Y))) IllegalAssignment();
if ((regionT(Y)=scoped) and (regionT(X)<>scoped)) IllegalAssignment();
if ((regionT(Y)=scoped) and (not shared(Y) and (not nested(X, Y))) IllegalAssignment();
```

Figure 1: Write barrier code detecting illegal assignment.

The header of the object must specify both the space and the region to which the object belongs. Then, when an object/array is created by executing the `new` (`new_quick`) or `newarray` (`newarray_quick`) bytecode, it is associated with the scope of both the active space and the active region. Local variables are also associated with the scope of the active region.

3 Collecting the local heap

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.

3.1 The local collector strategy

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]. This algorithm allows the interleaved execution of the collector and the application, which execution is synchronized by using write barriers to detect whether the application updates a pointer from a black object to a white one. Each root stack is processed root by root, and each object referenced by a root is inserted in a grey-list. If during this phase, the application treats to make a reference from a black object to a white one, the color of the referenced object is turned grey and the object is moved from the white-list to the grey-list, which is achieved by using write barriers.

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 [11] 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.

Since objects allocated within regions may contain references to objects within the local heap, a tracing-based GC (e.g., incremental or generational) 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 [5]. 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. When collecting a scope region because their reference-counter reaches zero, the root-list of the local GC is updated to remove all the objects in the region that are external roots for the GC.

3.2 Dealing with critical tasks

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. RTSJ [4] 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. A reference of a critical task to an object allocated in the heap causes the `MemoryAccessError()` exception, which we achieve by using *read barriers*. Read barriers occur upon all object accesses, which means upon executing both types of bytecodes: those which cause a *load reference* (i.e., the `getfield`, `aaload`, and `getstatic` bytecodes) and those causing a *store reference* (i.e., the `putfield`, `aastore`, and `putstatic` bytecodes). Note that read barriers are strictly necessary only when using read barrier-based collectors (i.e., incremental copying). When using a non-read barrier-based collector (i.e., mark-and-sweep, generational, 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. Then, the restriction on critical tasks can be reduced to write barrier checks. And, the `MemoryAccessError()` exception, which raises when a critical task attempts to 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 [5].

4 Related works

Unlike traditional JVMs, in our proposed solution, the applications run concurrently within a single JVM instance. In [6], we study the management of resource consumption taking into account both real-time constraints and the available memory budget. To support the memory model of RTSJ, the JVM must check for the assignment rules before to execute an assignment statement. Our solution consists to check both violation spaces, and the imposed RTSJ access and assignment rules dynamically, just when executing an assignment statement. In order to do so, we introduce an extra code in all bytecodes causing an object assignment. The use of write barriers to detect illegal assignments was introduced in [7]. Another effort to partition the Java memory is described in [3]. In this model, the creation of a new heap is optional; the proposed interface allows us to create a new name-space which shares the system heap rather than creating a new one. In order to avoid malicious cross-reference between private heaps, this solution uses both read and write barriers. As our solution, the implementation of heap partitioning binds heaps to objects by adding a field in the object header.

In order to provide strong isolation between services, both to enforce security and to control resource

consumption, the solution proposed in [12] subdivides a physical machine into a set of fully isolated protection domains. As in our solution, each virtual machine is confined to a private namespace. In a way similar to the Java Os from Utah [2], in order to provide secure and controlled accesses, we limit direct sharing among applications. 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 allowing activities to communicate using the RMI, and forbidding all possible communication. Objects in the shared heap are not allowed to have pointers to objects in any user heap. Then, attempts to assign such pointers will result in an exception, which is enforced by using write barriers, as in our proposed solution.

In order to avoid the problem with enforced write barriers, the solution proposed in [10] modifies the Isolate API by introducing two objects called `Portal` and `DefereedPortal` to communication among isolates, and a security manager providing hierarchical access rights. In this model, a parent can grant and revoke the communications rights of its children. The solution presented in [9] modifies the semantics of concurrent programs by combining Hoare´s monitors with the Java RMI. While in a Java application all threads share the same objects, in this solution application consist of a set of sequential processes not sharing any kind of data. In order to share object among processes, this solution introduces the `SharedObject` class and the `Shareable` interface. This solution is similar to our proposed mechanism to share objects within the common memory.

5 Conclusions

This paper has presented a memory management design solution for extending the RTSJ specification to execute several activities concurrently in the same JVM. 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. Our approach does not necessarily performs better than others RTSJ-based approaches, more over the use of write barriers to support the memory spaces reduces the JVM performance. Full class loading is required only for the first application that loads a given class. To communicate two tasks of different activities, we present a *limited sharing* model based on the common immortal memory space. We have no tread yet the scheduling problem, but we consider that multiprogramming scheduling algorithm in RTSJ is an interesting future work.

References

[1] H.G. Baker. "The Treadmill: Real-Time Garbage Collection without Motion Sickness". 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] P. Bernadat, D. Lambright, D., and F. Travostino. "Towards a resource safe Java for service guarantees in uncooperative environments". In Proc.s of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications, 1998.

[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] 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. Kluwer Academic Publishers, Vol.26, 2004.

[6] M.T. Higuera, "Memory Management Design to the Concurrent Execution of RTSJ". Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), LNCS 2889, november, 2003.

[7] 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.

[8] Java Community Process. Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp, 2003.

[9] L. Mateu. "A Java Dialect Free of Data Races an Without Annotations". In Proc of the International Parallel and Distributed Procesing Simposium (IPDPS), IEEE , 2003.

[10] K. Palacz, G. Czajkowski, L. Daynès, and J. Vitek. "Incomunicado: Efficient Communication for Isolates". ACM OOPSLA. November 2002.

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

[12] A. 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.