Supporting Several Real-time Applications on the Java Platform*

M. Teresa Higuera-Toledano

Facultad Informática, Universidad Complutense de Madrid, 28040 Madrid Spain

Email: mthiguer@dacya.ucm.es

Abstract

In this paper, we adapt the Java memory management to an embedded system, e. g., wireless PDA executing multimedia applications. We consider the concurrent execution of several applications within a single JVM. 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 Garbage Collector (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. An existing hardware support allows us to improve the performance of our solution.

1 Introduction

This paper focuses on a memory management solution in order to divide/share the heap among different real-time applications accounting for relevant Java specifications: *(i)* the *Real-time Specification for Java* (RTSJ) [3], *(ii)* the application isolation API [7], currently under revision as JSR-001 and JSR-121 respectively, *(iii)* the KVM [9] targeting limited-resource and network connected devices, and *(iv)* the picoJava-II [8] microprocessor. 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. The paper is organized as follows. Section 2 presents the general guidelines of our solution design to execute concurrently several applications. Section 3 improves the performance of the proposed solution by using an existing hardware support. Section 4 sketches an overview of related work. Finally, Section 5 offers some conclusions.

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 The Application Isolation API

Running multiple applications within a single instance of the same JVM has the potential for improving the performance and scalability of the system by sharing code and data structures. The communication among two applications running within the same JVM can be lighter than communication by using the RMI. The application isolation API [7] 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. From the programmer point of view, starting an isolate is the same to starting a new JVM. The Isolate class allows us to create an isolate by specifying a class. The only requirement is that the specified class must be a Java application, i.e. must have the `main()` method.

* Founded by the Ministerio de Ciencia y Tecnología of Spain (CICYT); Grant Number TIC2003-1321.

0-7803-8735-X/05/$20.00©2005 IEEE

2.2 The RTSJ Memory Model

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 RTSJ [3].

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 immo rtal 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.

2.3 Introducing the Memory Space 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.

To facilitate code sharing, classes are stored within the common space (i.e., the `CommonSpace.instance()` object). In this way, all applications in the system access both code and data. Class variables, declared as `static` in Java 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. The solution is then to allocate also in the immortal memory of the application a copy of the static code. As in the solution given in [4], we maintain a copy of both class variables and class monitors for each application using the class, while maintaining only a single version of the class code. This solution requires modifying the class loader.

2.4 Dynamic Detection of Illegal Assignments

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 referenced object is within a protected space. For intra-spaces references, we must take into account the assignment rules imposed by RTSJ [3] (i.e., objects within the heap or an 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). Whereas for inter-spaces references, we raise an `illegalAssignment()` exception.

- Treatment B: the referenced object is within the common space. It is allowed and nothing needs to happen.

In Figure 1, we can see the write barrier pseudo-code that we must 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. 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 [6].

```
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 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 both the active region scope and the active space scope.

3 Using Hardware Support

In this section, we first present an overview of the write barrier hardware support that the picoJava-II microprocessor [8] provides. Next, we introduce a hardware-based solution to improve the write barrier performance of memory regions.

3.1 The picoJava-II write barrier

Upon each instruction execution, the picoJava-II core checks for conditions that may cause a trap. From the standpoint of GC, this microprocessor checks for the occurrence of write barriers, and notifies them using the **`gc_notify`** trap. This trap is triggered under certain conditions when assigning to an object's field an object reference (i.e., when executing bytecodes requiring write barriers). The conditions that generate the **`gc_notify`** trap are governed by the values of the `GC_CONFIG` and the `PSR` registers. The `GC_CONFIG` register governs two types of write-barrier mechanism: page-based and reference-based. Whereas the reference-based write barriers are used to implement incremental collectors, the page-based barrier mechanism was designed specifically to assist generational collectors based on the train-based algorithm.

3.2 Supporting Memory Spaces

Our solution uses the picoJava-II paged-based mechanism to detect references across different spaces by mapping each space in a car. If the `GCE` bit of the `PSR` register is set, then page-based write barriers are enable. The object reference in picoJava-II has 4 fields: `GC_TAG`, `ADDRESS`, `X`, and `H`. The `ADDRESS` field (bits <29:2>) of the reference always points to the location of the object header. In the `GC_CONFIG` register, the `TRAIN_MASK` field (bits <31:21>) allows us to know whether both objects in an assignment X and Y belong to the same train, whereas the `CAR_MASK` field (bits <20:16>) detects whether they belong to different cars (see Figure 2). If, for example, we initialize the `REGION_MASK` field as **0000000000**, and the `CAR_MASK` field as **11111**, we have only a train divided in 32 cars (spaces), each one divided in pages of 16 Kbytes.

```
if (PSR.GCE = 1) then
  if ((X<29:19>&GC_CONFIG<31:21>)=(Y<29:19>& GC_CONFIG<31:21>))
and
    ((X<18:14>&GC_CONFIG<20:16>)<>(Y<18:14>&GC_CONFIG<20:16>))
then gc_notify trap
```

Figure 2: Page-based write barrier mechanism.

The page-based mechanism avoids us to execute the write barrier code when both objects X and Y belongs to the same space (i.e., for intra-space assignments). Then, write barriers are executed only for references across spaces (i.e., for inter-space assignments). Note that spatial locality property means that intra-space assignments are more frequent than inter-space assignments. An application makes an inter-space assignment for two proposes: to communicate to another application by sharing an object within the common space or to violate the protected space of another application. Since both communication among applications and violation spaces are infrequent, the improvement introduced in the performance of our solution is important. Figure 3 shows the associated exception routine to the **gc_notify** trap, which must be executed for inter-space assignments.

```
gc_notify_trap_code
      if (space(Y) <> common) IllegalAssignment();
         priv_ret_form_trap;
```

Figure 3: Detecting illegal assignments across spaces.

3.3 Implementation Details

We have limited to 8 the number of memory spaces. We consider that spaces are paged, and the page size is 64 Kbytes. Also, the maximum number of pages that a space can hold has been limited to 32. We consider further that a maximum of 64 tasks can reference objects in the same scoped region. To limit the worst case for write barriers execution time, we must to limit the number of scoped nested levels. Since we must to take into account the *single-parent rule* of scoped regions [3], this means that we must limit the maximum number of scoped regions that an application can hold.

We have fixed this limit to 30, which allows us to support the region by using 5 bits, and the region stack of each task in 10 words. With this implementation, the overhead introduced in the KVM [9] to evaluate a condition of the write barrier test is about 17% per assignment. Instead of using the SPECjvm98 benchmark, which is not compatible with the KVM,we use an artificial collector benchmark. This is an adaptation made by Hans Boehm from the Ellis and Kovac benchmark.

This benchmark executes 262 millions bytecodes and allocates 408 MBytes. Since the number of bytecodes that perform a write barrier test is 15 millions, we conclude that 5% of executed bytecodes perform a write barrier test. Where 92.4% of the references are to object variables (i.e., `aastore`: 6.6%, `putfield`: 39.6%, and `putfield_fast`: 46.2%), and 7.6% to class variables (i.e., `putstatic`: 6.6%, and `putstatic_fast`: 1%). Note that our solution allocates the object variables within the heap or a scoped region, and the class variables within an immortal region.

3.4 Memory footprint

In order to adapt the KVM objects to the picoJava-II microprocessor, we add a word to the object header of the KVM. The added word includes the following fields: `REGION_TYPE` <31:30> (`GC_TAG` in picoJava-II), the `REGION_ID` <29:25> and the `SPACE_ID` <18:14> (`CAR_ADDRESS` in picoJava-II). Where the `REGION_ID` and the `SPACE_ID` fields specify respectively the region and space to which the object belongs, and the `REGION_TYPE` specifies the region type (e.g., 00 for the heap, 01 for immortal, 10 for scoped non-shared, and 11 for scoped shared). This increases a word per object the memory consumption. Alternatively, we can modify the original header format of KVM objects (i.e., `SIZE` <31:8>, `TYPE` <7:2>, `MARK_BIT` <1>, and `STATIC_BIT` <0>) to support the identification of both the space and the region to which the object belongs, and also the type of the region (i.e., `REGION_TYPE` <31:30>, `SIZE_H` <29:19>, `REGION_ID` <18:14>, `SIZE_L` <13:8>, `TYPE` <7:2>, `MARK_BIT` <1>, and `STATIC_BIT` <0>). Note that the maximum size of the object has been reduced from 16 Mbytes to 1 Kbytes; given the small average object size that the specJVM [16] applications present (i.e., about 32 Bytes), we optimize for small objects.

We maintain a *region-structure* of 2 words for each MR object in the system with the following format: `REGION_TYPE` <31:30>, `REGION_ID` <29:25>, `OUTER_REGION_ID` <24:20>, `REFERENCE_COUNTER` <19:14>, `SIZE` <13:8>, `SPACE_ID` <7:5>, and `PAGE` <4:0>. Where the `REFERENCE_COUNTER`, the `SIZE`, and the `PAGE` fields allow us to know respectively: the number of tasks that can allocate or reference objects in the region, the size hold by the region in bytes, and the page which support the region. The region-structure increases the memory footprint as maximum of 128 Bytes. Note that these region-structures forms a *scope-tree* [6] where the heap is the root and immortal regions are not included.

5 Related Works

Unlike traditional JVMs, in our proposed solution, the applications run concurrently within a single JVM instance. In [5], 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. In order to do so, we introduce an extra code in all bytecodes causing an object assignment.

Another effort to partition the Java memory is described in [2]. 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 [10] subdivide 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 [1], 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 allow activities to communicate using the RMI, and forbidding all possible communication.

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. To facilitate code sharing, classes are stored in the immortal common space. The partition of memory allows us to invoke several collectors concurrently, where the reclamation rate can be different for each application. Regarding our software-based solution, we found only two problems: the high overhead that introduces the dynamic check of illegal assignment, and that this overhead must be bounded by limiting the nested scoped levels.

Our solution, to improve the performance of memory management, partly addresses the use of hardware aid by exploiting existing hardware support for Java. This solution is efficient, but not very flexible, because we must configure the system to determine the virtual region memory map, which can be unpractical for RTSJ classes dealing with I/O mapped memory (e.g., `ImmortalPhysicalMemory`). Also requires the size of the space to be multiple of the car size and the size of the region to be multiple of the page size, which may introduce internal fragmentation. These problems can be avoided by using the header of the object in the write barrier mechanism instead of the object reference.

References

[1] 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.

[2] P. Bernadat, D. Lambright, D., and F. Travostino. "Towards a resource safe Java for service guarantees in uncooperative environments". In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications,* 1998.

[3] 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

[4] 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.

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

[6] M.T. Higuera-Toledano 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.

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

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

[9] Sun Microsystems. "KVM Technical Specification". Technical Report. Java Community Process, May 2000. http://java.sun.com

[10] 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.