Hardware Support for Detecting Illegal References in a Multiapplication Real-Time Java Environment

M. TERESA HIGUERA-TOLEDANO
Universidad Complutense de Madrid

Our objective is to adapt the Java memory management to an embedded system, e.g., a wireless PDA executing concurrent multimedia applications within a single JVM. This paper provides software, and hardware-based solutions detecting both illegal references across the application memory spaces and dangling pointers within an application space. We give an approach to divide/share the memory among the applications executing concurrently in the system. We introduce and define application-specific memory, building upon the real-time specification for Java (RTSJ) from the real-time Java expert group. The memory model used in RTSJ imposes strict rules for assignment between memory areas, preventing the creation of dangling pointers, and thus maintaining the pointer safety of Java. Our implementation solution to ensure the checking of these rules before each assignment inserts write barriers that use a stack-based algorithm. This solution adversely affects both the performance and predictability of the RTSJ applications, which can be improved by using an existing hardware support.

Categories and Subject Descriptors: Programming Languages, Processors-memory management, garbage collection, run-time environments []

General Terms: Languages, design, performance, algorithms

Additional Key Words and Phrases: Write barriers, memory management, garbage collection

1. INTRODUCTION

The demand for multimedia services in embedded real-time systems, such as wireless personal digital assistants (PDAs), is increasing. The use of PDAs is foreseen to outrun the use of PCs in the near future. However, for this to actually happen, there is still the need to devise adequate software and hardware platforms that will not overly restrict the applications that are supported. In

This research was supported by Consejería de Educación de Comunidad de Madrid, Fondo Europeo de Desarrollo Regional (FEDER) and Fondo Social Europeo (FSE), through BIOGRIDNET Research Program S-0505/TIC/000101, and by Ministerio de Educación y Ciencia, through the research grant TIC2003-01321.
Author's Address: M. Teresa Higuera-Toledano, Faculted Informática, Universidad Complutense de Madrid, Ciudad Universitaria, 28040 Madrid Sain; email: mthiguer@dacya.ucm.es.

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 new applications that extend the capabilities of the host system.

The ideal candidate for providing an open environment is Java, which appears as a major player in the area of embedded software environment and allows us to obtain portable code, which can possibly be dynamically downloaded. Standard Java has some shortcomings regarding the target device that have been solved by extending the Java API to meet the requirements appertained to embedded real-time software [Bernadat et al. 1998], such as real-time scheduling and predictable memory. These changes will lead to even greater need for executing multiple applications in parallel on the same JVM (e.g., [Baker 1991] and [Czajkowki 2000]). 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 RMI.

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: (1) the application isolation API [Java Community Process 2003], (2) the real-time specification for Java (RTSJ) [Bollella et al. 2002], currently under revision as JSR-121 and JSR-001 respectively, (3) the KVM [Sun Microsystems 2000] targeting limited-resource and network connected devices, and (4) the picoJava-II [Sun Microsystems 2000] microprocessor architecture.

1.1 Background

The main issue in delivering allocation is related for providing isolation guarantees that ensure that an application will not be disrupted by failure or misbehavior caused by another application running in the system. When executing multiple applications concurrently, if one application consumes all the available memory, the other applications get starved. One way to avoid this problem is to divide the memory among applications running in the system, giving each application a separate garbage collectable area. The application isolation API [Java Community Process 2003] 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 object graphs and sharing objects among two different isolates is forbidden. From the programmer point of view, starting an isolate is the same as starting a new JVM. Isolates are created from standard Java applications. The only requirement is that the specified class must be a Java application (i.e., must have the main() method). As the Java Runtime class, the Isolate class provides methods to terminate the execution of isolates (i.e., the exit() and halt() methods). The termination of an isolate is guaranteed to leave the system in a consistent state.

The partition of the heap in separate subheaps allows us to invoke several collectors concurrently; having a collector per subheap that is customized according to the behavior of the embedded application minimizes the latency time to preempt a local collector from the CPU when a high-priority task from another application arrives, and distributes the collector overhead among applications. 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 alternative approach to the GC is to use memory regions within which both allocation and deallocation are customized and also space locality is improved. Application of these two implicit strategies has been studied in the context of Java, which are combined in RTSJ [Bollella et al. 2002].

The MemoryArea abstract class supports the region paradigm in the RTSJ specification through the following three kinds of regions: (1) immortal memory, supported by the ImmortalMemory and the ImmortalPhysicalMemory classes, that contains objects whose life ends only when the JVM terminates; (2) (nested) scoped memory, supported by the ScopedMemory abstract class, that enables grouping objects having well-defined lifetimes; and (3) the conventional heap, supported by the HeapMemory class. An application can allocate memory on the system heap, the immortal system memory region, several scoped memory regions, and several immortal regions associated with physical characteristics. When entering a new scope by the enter() method of the instance or by starting a new task (i.e., by creating an instance of RealtimeThread or NonHeapRealtimeThread), whose constructors were given a memory region. In the second case, an object created by the task is allocated within memory associated with this scope. When the scope is exited by returning from the enter() method, all objects will allocate within the memory associated with the enclosing scope (i.e., the nested outer scope). Allocations outside the active region can be performed by the newInstance() or the newArray() methods.

As an example, the code of Figure 1 shows a thread called myThread (line 21) allocating an object A within the heap (line 6), a real-time thread called myTask (line 22) allocating an object B within the scoped memory region called myVTRegion (line 12), an object C within the scoped MR called myLTRegion (line 14), which is inner to myVTRegion (line 13), and another object D within the immortal region (line 15). The scoped memory region myVTRegion was created with a size of 2 KB and can grow to up 4 KB (line 20), whereas the scoped memory region myLTRegion was created with a size of 1 KB and cannot grow (line 26). The MemoryParameters object limits the amount of memory that myTask may allocate to 3 KB for the myVTRegion memory region, to 0 KB for the immortal memory region, and to 1 KB/sec for the heap (line 23).

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. A scoped region gets collected as a whole once it is no longer used. The lifetime of objects allocated in scoped regions is governed by the control flow. Strict assignment rules placed on assignments to or from memory regions prevent the creation of

```
1:  import javax.realtime;
2:  class RegionUseExample {
3:
4:    class R1 implements  Runnable {
5:       public void run() {
6:         myObject A = new myObject();
7:                ..................... // do some stuff
8:     }
9:
10:    class R2 implements  Runnable {
11:      public void run() {
12:        myObject B = new myObject();
13:        myLTRegion.enter();
14:        myObject C = myLTRegion.newInstance(myObject());    ;
15:        myObject D = ImmortalMemory.instance().newInstance(myObject());
16         ..................... // do some stuff
17:     }
18:
19:     public static void main (String[] args)  {
20:       ScopedMemory myVTRegion =  new VTMemory(2*1024, 4*1024);
21:       Thread myThread = new Thread(new R1());
22:       RealtimeThread myTask = new RealtimeThread(....., .....,
23:                               new MemoryParameters(3*1024, 0, 1024),
24:                               myVTRegion,  .....,
25:                               new R2());
26:       ScopedMemory myLTRegion = new LTMemory(1024, 1024);
27:         myThread.start();
28:         myTask.start();
29:       }
30:     }
```

Fig. 1. Using memory regions in RTSJ.

Table I. Assignment Rules in RTSJ

	Reference		
	Heap	Immortal	Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Same or outer

dangling pointers (see Table I). These rules avoid references from objects within the heap or an immortal memory to objects within a scoped region, and from objects within a scoped region to objects within another scoped region that is nonouter.

The JVM must check for the assignment rules before executing an assignment statement and throw an illegalAssignment() exception, if they are violated. This check includes the possibility of static analysis of the application logic [Bernadat et al. 1998]. In the above example, illegal references are the following two cases: from a field of an object within the heap or an immortal region to an object within either myVTRegion or myLTRegion, or from a field of an object within myVTRegion to an object within myLTRegion (see Figure 2). The following assignment statement can then cause dangling pointers and, as a consequence, are illegal references: A.field = B, A.field = C, D.field = B, D.field = C, and B.field = C. On the other hand, there is no problem with all other possible assignments.

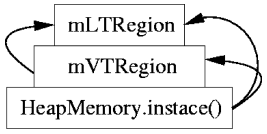

Fig. 2. Illegal references in the Figure 1 program example.

Note that the assignment rules in RTSJ avoid pointers from an object to another object within a region with a live-time potentially shorter than the region of the referenced object. Pointers from an object within the ImmortalMemory instance to objects within the HeapMemory instance and vice versa are always allowed.

1.2 Related Work

The design of real-time Java applications is already a problem. In particular, memory management must be carefully designed to be compliant with real-time constraints. In [Issarny et al. 2000], we present a study of the requirements for embedded software environment aimed at wireless PDAs. These requirements have been addressed through a middleware platform where, unlike traditional JVMs, several applications run concurrently within a single JVM instance. For the collection model, the idea that this paper presents, which consists of a dedicated GC running within each application and a global real-time GC to reclaim the objects created by the environment, as well as those shared among applications, is too complicated and inefficient. Implicit garbage collection comes along with overhead regarding both execution time and memory consumption; two levels of garbage collection makes it very poorly suited for small-sized embedded real-time systems.

In [Higuera-Toledano 2004], we continue with the idea of running several real-time Java applications within the same JVM by extending the memory-management model of RTSJ to offer multiprocess execution. In the proposed solution, there is a memory space accessible by all applications in the system, which allows interprocess communication by using both the communication model of standard Java (based on shared variables and monitors), and the specific classes of RTSJ that provide communication among real-time tasks and threads. We have a distinct GC per application, but different from the scheme presented in [Issarny et al. 2000]. Shared objects among applications are not collected, which simplifies the resource management. We take this solution as a trade-off between a more general solution, such as allowing activities to communicate using RMI, and forbidding all possible communication. To facilitate code sharing, classes are stored within the same common memory space where shared objects are allocated.

In RTSJ, the life of objects allocated in scoped regions is governed by the control flow. To maintain the safety of Java and avoid dangling references, the JVM must check for the assignment rules before executing an assignment statement: objects within a scoped region can only be referenced by objects from the same region or within an inner region, and objects within immortal regions or within the heap cannot reference an object allocated in a scoped region.

Our solution introduces extra code for all bytecode instructions that operate on references within other objects (or arrays) [Higuera-Toledano et al. 2000]. This extra code, normally called *write barrier*, must be executed before updating the object reference (i.e., when an instruction causes a reference from an X object to a Y object). The write barrier code explores a scope stack, from the scope of X (the active region) down to the scope of Y (an outer region). If the scope of Y is not found in the stack (the bottom of the stack is reached), this is notified by throwing an exception.

The aforementioned solution presents two problems: the high overhead introduced by the dynamic check of illegal assignment for scoped memory regions, and that this overhead must be bounded by limiting the nested scoped levels. In order to reduce the execution time taken to check illegal references caused by assignments among scoped regions, we present a solution to improve the performance of write barriers [Higuera-Toledano 2003], which resorts to hardware aid by exploiting existing hardware support for Java (i.e., the picoJava-II microprocessor), and also combines with a specialized hardware, which allows us to limit the nested scoped levels and reduces to zero the execution time to explore the scope stack.

The picoJava-II microprocessor [Sun Microsystems 2000] has been discontinued by Sun some time ago. This microprocessor was never released as a product by Sun, but it has been licensed to several companies (e.g., Fujitsu, IBM, LG, and NEC), which also did not produce a chip. However, picoJava-II is the Java microprocessor most often cited in research papers and it still is a perfectly fine platform to explore new approaches. The aJile's JEMCore [Hardin 2001] is a direct-execution Java processor that is available as both an IP core and a stand-alone processor. It is based on the JEM2 Java chip developed by Rockwell–Collins, which decided not to sell the chip on the open market. Instead, it licensed the design exclusively to aJile Systems. Komodo [Uhrig et al. 2002] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller. JOP [Shoeberl 2005] is a hardware time-predictable Java platform for embedded real-time systems, with small design that fits into a low-cost FPGA. It is also a working processor, not merely a proposed architecture.

The contribution of our work in this article comes from the adaptation and integration of the relevant solutions obtained in our previous work, in the context of the RTSJ, based on the analysis of the parameters that are the most influential in memory–management performance. We support several applications within the same JVM by introducing memory spaces in RTSJ. We resolve illegal interregion assignment in RTSJ by using hardware support in order to improve the write-barrier execution time. In addition, we have discussed the time and memory overhead introduced by the resulting memory-management solution within the KVM.

Another effort to partition the Java memory is described in [Bernadat et al. 1998]. In this model, the creation of a new heap is optional; the proposed interface allows creating a new name-space, which shares the system heap, rather than creating a new one. In order to avoid malicious cross-references between private heaps, this solution uses both read and writes barriers. The

implementation of heap partitioning binds heaps to objects by adding a field in the object header. The Java Os from Utah [Back et al. 1998] provides secure and controlled accesses, by limiting direct sharing among applications. When two activities want to communicate, they must share an object residing in the common heap. Objects in the shared heap are not allowed to have pointers to objects in any user heap. Attempts to assign to 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 [Palacz et al. 2002] modifies the Isolate API by introducing two objects called Portal and DeferreedPortal to communicate between Isolates, and a security manager providing hierarchical access rights. In this model, a parent can grant and revoke the communication rights of its children. A different solution to improve resource utilization consists of sharing libraries among JVMs [Wong et al. 2003]. In this case, the challenge is to determine what to share and how to share it in order to decrease both start-up time and memory footprint without compromising the robustness of the system.

The solution proposed in [Whitaker et al. 2002] provides strong isolation between services, both to enforce security and to control resource consumption. In order to do that, this solution subdivides a physical machine into a set of fully isolated protection domains. Each virtual machine is then confined to a private namespace.

A reconfigurable virtual machine supporting multiple user environments with varying degrees of criticality and privilege has been presented in [Jensen et al. 1999]. This architecture provides hardware-enforced guarantees of resource separation and is based on the JEM-1 Java-based microprocessor. Hardware-based Java platforms (e.g., [Hardin 2001]) provide efficient support for bytecode execution, hard real-time, and also safe and secure multiple virtual machine execution.

1.3 Paper Organization

The rest of this paper is organized as follows. We first present our basic approach, which includes write barriers to detect whether an application attempts to create an illegal assignment to references inside or outside its memory space, and a description of how memory spaces and scoped regions are supported (Section 2). Next, we propose a hardware-based solution to improve the performance of write barriers, and a specialized hardware that makes the time-cost to detect illegal assignments across scoped regions negligible (Section 3). We present basic modifications to the Java VM in order to make it aware of memory regions, and evaluate the overhead introduced by write barriers in our solution by instrumenting the KVM (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 multiprocess execution. In the proposed solution, some memory objects are accessible by all applications in the system. This allows interprocess

communication by using both the communication model of Java based on shared variables and monitors as well as the classes that the RTSJ specification provides to communicate real-time tasks and non-real-time threads.

2.1 Extending the Memory Class Hierarchy

In order to obtain multiprocess execution, we introduce the abstract class `MemorySpace` 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 system initialization time, and is a resource shared among all applications in the system. In contrast, a new `ProtectedMemory` object is created when estabalishing a new application and is a local resource protected from accesses of all 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.

2.2 Sharing Common Resources

To facilitate code sharing, classes are stored within the common space (i.e., the `CommonMemory.instance()` object). Thus, all applications in the system can access both code and data (i.e., *class variables*), of all classes. However, 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. Note that in order to isolate the class variables of an application from accesses of other applications, we maintain a copy of class variables for those application using they. Since the goal is isolation, not replication, we must not maintain copy coherence. Thus, when an application needs a new class, the class loader checks if the class already exists within the common space:

- If NO, the class is loaded in the common space and copies of all class variables are allocate within the application local heap,
- if YES, copies of all class variables are allocated within the application local heap.

The same problem arises with *class monitors* (i.e., shared code related to synchronization); these methods are declared in `Java as static synchronize`. 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 to allocate a copy of the static code within the immortal memory of

the application. As in the solution given in [Czajkowki et al. 2000], 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 Dynamic Detection of Illegal Assignments

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

- Case A: the referenced object is within the common space. The assignment is allowed and nothing needs to happen.
- Case B: the referenced object is within a protected space. For intraspace references, we must take into account the assignment rules imposed by RTSJ (see Table I). For interspace references, we raise a `SpaceViolated()` exception.

In order to detect illegal assignments to scoped regions, every thread is associated with a region-stack containing all scoped memory regions that the thread can hold. Every scoped region is associated with a reference count that keeps track of the use of the region by tasks. The memory region at the top of the stack is the active region for the task, whereas the memory region at the bottom of the stack is the outermost scoped region for the task. The default active region is the heap. When a task does not use any scoped region, the region-stack is empty and the active region is the heap or an immortal memory region. Checking nested regions requires two steps:

1. In a first step, the region-stack is explored top down to find the memory region to which the X object belongs. If it is not found, a `MemoryAccessError()` exception[1] is thrown.
2. A second step explores the region-stack again, starting at the region that contains the X object, and the objective is to find the region that contains the Y object (i.e., the region to which Y belongs must be outer to the region to which X belongs). If the scoped region of Y is found, the assignment is allowed.

Take as an example, the code given in Figure 1. Where `B` array is allocated within the `myVTRegion` scoped region (line 12) and `C` array is allocated within the `mLTRegion` scoped region (line 14), and `mLTRegion` scoped region is inner to `myVTRegion` scoped region. Note that the `myTask` real-time thread entered `mLTRegion` when `myVTRegion` was the active region (line 13). The algorithm works as follows:

- Assignment `C[i]=B[j]` returns `true` (see Figure 3a).
- Assignment `B[i]=C[j]` raises the `IllegalAsignment()` exception (see Figure 3b).

[1]This exception is thrown upon any attempt to refer to an object in an inaccessible MemoryArea object.

a. *Allowed: Y is outer to X.* *b. Illegal: Y is inner to X.*

Fig. 3. First (look up X) and second (look up Y) explorations of the region-stack.

```
if ((spaceType(Y)=protected)  and (spaceName(X)<>spaceName(Y))) SpaceViolated();

if ((regionType(Y)=scoped)  and (not nestedRegions(X, Y)) IllegalAssignment();
```

Fig. 4. Write-barrier code detecting illegal assignment.

Figure 4 presents the write-barrier pseudocode that is introduced in the interpretation of the putfield, aastore, and putstatic bytecodes. The spaceType() function returns common or protected depending on the type of space to which the object parameter belongs. The spaceName() function returns the space identifier to which the object parameter is allocated. The regionType() function returns heap, immortal, or scoped, depending on the type of the region to which the object belongs. The nestedRegions(X,Y) function returns true if the scope of the region that contains the Y object is the same or outer to the scope of the region that contains X.

The header of the object must specify both the space and the region to which the object belongs. When an object/array is created by executing the new or newarray bytecodes, it is then 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

To efficiently implement a generational or an incremental garbage collector, picoJava offers hardware support for write barriers through memory segments. The hardware checks all stores of an object reference if this reference points to a different segment (compared to the store address). In this case, a trap is generated and the garbage collector can take the appropriate action. Two additional reserved bits in the object reference can be used for a write-barrier trap. In this section, we use the write-barrier hardware support that the picoJava-II microprocessor provides. Next, we introduce a specialized hardware to improve the write-barrier performance of scope memory regions.

3.1 Supporting Memory Spaces

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 indicates them using the gc_notify trap. This trap is triggered under certain conditions when assigning an object reference to an object's field (i.e., when executing bytecodes requiring write barriers),

```
if ((X<29:19> & GC_CONFIG<31:12>) = (Y<29:19> & GC_CONFIG<31:12>))
        and ((X<18:14> & GC_CONFIG<20:16>) <> (Y<18:14> & GC_CONFIG<26:16>))
then gc_notify trap
```

Fig. 5. Page-based write-barrier mechanism.

and governs two types of write-barrier mechanism: page- and reference-based. Whereas the reference-based write barriers are used to implement incremental collectors, the page-based write barriers are used to implement generational collectors.

The page-based barrier mechanism of the picoJava-II was designed specifically to assist generational collectors based on the *train* algorithm [Wilson and Johnston 1993], which divides the object space into a number of fixed blocks called *cars*, and arranges the cars into disjoint sets (*trains*). This algorithm tracks references across different cars within the same train. The conditions that generate the gc_notify trap are governed by the values of the GC_CONFIG register and the *page-based status register* (PSR). 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. If the *garbage collection enable* (GCE) bit of the PSR register is set, then page-based write barriers are enabled (see Figure 5).

In order to reduce the cost of object relocation, the strategy adopted by the *Sun* JDK and SDK is to add a nonmoving handle (i.e., a reference) to each object. Thus, each object has a reference that points to the location of the object header (see Figure 6).

When an object is relocated, its handle is updated. Thus, relocating objects is transparent to the application program, which always accesses objects using their nonmoving reference. In picoJava-II, the ADDRESS field (bits <29:2>) of the object reference always points to the location of the object header, and is divided in two subfields: the TRAIN field (bits <29:19>) and the CAR field (bits <18:14>). The, page-based write barriers in picoJava-II uses the object reference, not the object header, which makes a physical division of the object memory address.

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 pagesof 16 KB (see Figure 7). Our solution uses the picoJava-II page-based mechanism to detect references across different memory spaces by configuring only one train and mapping each space in a car (i.e., we have as maximum 32 memory spaces divided in pages of 16 KB and each memory space can hold 1012).

The page-based mechanism allows us to save the execution of the write-barrier code when both objects X and Y belong to the same space (i.e., for intraspace assignments). Write barriers only need to be executed for references across spaces (i.e., for interspace assignments). Figure 8 shows the exception code associated with the gc_notify trap, which must be executed for interspace assignments. Note that as consequence of spatial locality property, intraspace

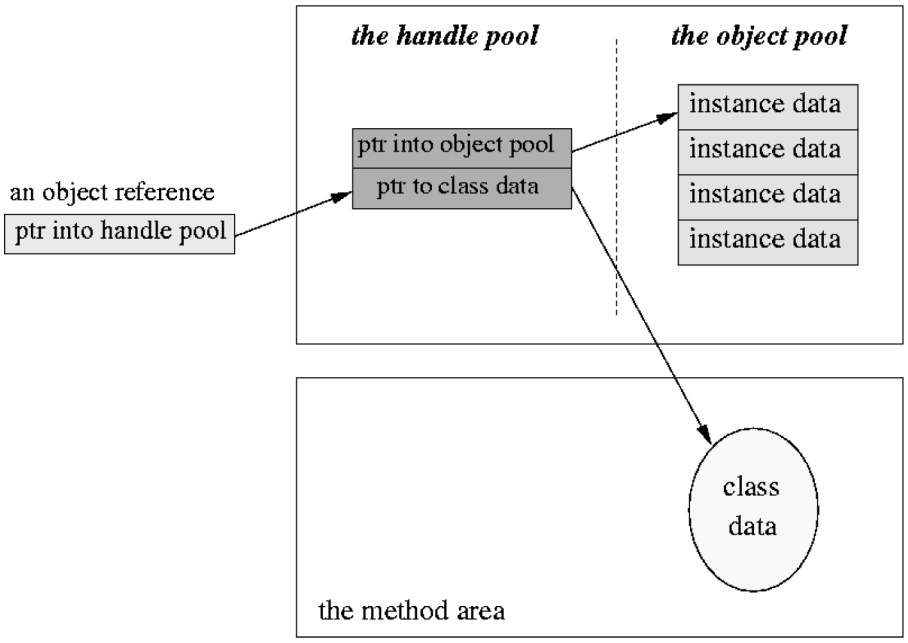

Fig. 6. Object structure with a nonmoving handler.

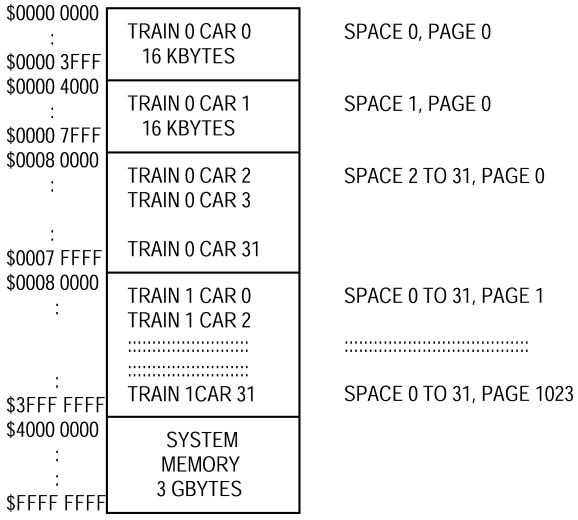

Fig. 7. Heap divided in trains and cars: CAR_MASK with 11111.

assignments are more frequent than interspace assignments. An application makes an interspace assignment for two purposes: to communicate with another application by sharing an object within the common space or to violate the protected space of another application. Since we consider that both communication among applications and violation spaces are infrequent, the performance improvement introduced by our solution is relevant.

```
gc_notify
        if (spaceType(Y) = protected) SpaceViolated();
        priv_ret_form_trap;
```

Fig. 8. Detecting illegal assignments across spaces.

```
if PSR.GCE = 1 then
        gc_index := (X<31 :30> << 2) | Y<31 :30>
        write_barrier_bit := (GC_CONFIG >> gc_index) | 00000001)
        if (write_barier_bit = 1) then gc_notify trap
```

Fig. 9. Reference-based write barriers.

This solution is efficient, but not very flexible, because we must configure the system to determine the virtual space memory map, which can be unpractical for RTSJ classes dealing with I/O mapped memory (e.g., ImmortalPhysicalMemory). It also requires the size of the memory space to be a multiple of the car size, which may introduce internal fragmentation. In order to avoid these problems, we can add a word to the header of the object to support both the TRAIN and CAR fields (i.e., the memory space to which the object belongs), and modify the page-based hardware support of picoJava-II to use the header of the object in the write-barrier mechanism instead of the object reference. Thus the memory division in spaces is logical, not physical.

3.2 Supporting Memory Regions

The reference-based write barriers of picoJava-II can be used to implement incremental collectors based on the *tricolor* algorithm [Baker 1991]. In this terminology, a *white* object means either garbage or that it has not been reached yet, a *grey* object means that it has been marked as reachable, but its contents have not yet been scanned; once all the pointers of the grey object are marked as reachable (grey), it is colored *black*. In order to synchronize the application and the collector, the tricolor invariant is introduced: *no black objects have references to* white *objects*. The application must then preserve this invariant by changing the colors of the nodes affected, if necessary.

The picoJava-II reference-based write barriers avoid executing write-barrier code when the object assignment does not attempt to violate the tricolor invariant. This mechanism also, allows us to improve the performance of both the collector and the application by disabling execution of write barriers when disabling the collector (see Figure 9). In picoJava-II, the GC_TAG field (bits <31:30>) of the object reference supports the reference-based write barriers supporting an incremental collector, which traps when a white object is written into a black object (e.g., the GC_TAG field for black objects is set to 11 and for white objects is 00).

In order to use this hardware mechanism, we use the GC_TAG field to support the type of the memory region to which the object belongs (e.g., 00 for the heap, 01 for the immortal region, 10 for immortal physical regions, and 11

```
gc_notify
        if (spaceType(Y) = protected) SpaceViolated();
        if (not nestedRegions(X, Y)) IllegalAssignment();
        priv_ret_form_trap;
```

Fig. 10. Write-barrier code detecting illegal interregion assignment.

```
gc_notify
        if (spaceType(Y) = protected) SpaceViolated();
        enable_mr_notifr;              //set PSR.MRE
        nop;
        disable_mr-notify              //unset PSR.MRE
        priv_ret_form_trap;
```

Fig. 11. Code for the gc_notify trap.

for scoped regions) and we configure the GC_CONFIG.WB_VECTOR to enable the gc_notify exception for references from an object within a nonscoped region to an object within a scoped one. In addition, we program the associated exception handler to execute the write-barrier code needed to detect the illegal assignments for scoped regions (see Figure 10).

This solution avoids the execution of write barriers for valid interregion references that do not require any action (i.e., for references to an object within an immortal region or within the heap). In order to improve the performance of our solution, we are interested in reducing the execution time taken to check illegal references caused by assignments among scoped memory regions, which depend on the region stack size. In Higuera-Toledano [2003], we have introduced hardware for executing the nestedRegion(X,Y) function, which supports the region stack of the active task in an associative memory.

Similar to the write-barriers mechanism of picoJava-II, our proposed hardware checks for the occurrence of illegal assignments across scoped regions and indicates them using the mr_notify trap. This hardware basically consists of an associative memory containing a mark bit per entry, called *Present-Bit* (P), which indicates whether the corresponding entry must be considered as an element of the region stack. If P is set, the corresponding entry is an element of the region stack of the active task. If it is reset, the corresponding entry is considered empty. This bit is used in the first step of the algorithm, when the region of object X is looked up in the region stack. Each entry contains also a valid bit called *Valid-Bit* (V), which indicates whether the corresponding entry must be considered at the second step of this algorithm, when the region of object Y is looked up in the region stack. When V is set, the entry contains an outer region to the region that contains X.

The condition under which the mr_notify trap is generated can be governed by a *reserved* bit in the PSR register, that we call *memory region enable* (MRE).

Each time a task is scheduled for execution, all entries of the SCOPED_STACK must be configured with the scoped region stack that the scheduled

task can access. This configuration introduces some overhead at context-switch time. In order to access/configure the SCOPED_STACK memory, we extend the picoJava-II instruction set by introducing the priv_read_scoped_stack and priv_write_scoped_stack bytecodes, whose operands are the entry index and a memory address to load/store the region identifier.

4. EVALUATING THE OVERHEAD

In this section, we estimate the write-barrier overhead introduced by the proposed solutions. We are interested in fixing an upper bound for the overhead introduced by: (1) checking an illegal inter-space assignment, (2) checking an illegal inter-region assignment, and (3) exploring the region stack. To quantify the overhead of write barrier, two measures are combined: the number of events (*Events*) and the measured cost of the event (*Cost*).

4.1 Quantifying Events

All objects created by a JVM are allocated within the heap (i.e., dynamic memory that in RTSJ may be either the heap or another memory region and, in our extended RTSJ, may be also the common space); only primitive types are allocated in the run-time stack. In most applications of the SPECjvm98 benchmark suite [Wong et al. 2003], less than one-half (45%) of the references are to objects within the heap rather than primitive types (e.g., bytes or integers), which the other one-half is to either the *Java* or *the native stack*. About 35% of the total executed bytecodes requires an object reference, where typically 70% is for load operations and 30% for store operations [Kin and Hsu 2000]. Then, 15% (0.45*0.35) of the bytecodes reference an object within the heap and 30% of these bytecodes make assignment operations. References to objects outside the stack require write barriers, whose probability is 5% (0.15*0.30).

Instead of using the SPECjvm98 benchmark, which is not compatible with the KVM, we use an artificial collector benchmark. In this benchmark, two data structures of the same size are kept around during the entire process: a tree containing many pointers and a large array containing integers. This benchmark executes 262 millions bytecodes and allocates 408 MB. 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 (6.6% by aastore 39.6% by putfield, and 46.2% by putfield_fast: 46.2%), and 7.6% to class variables (6.6% by putstatic and 1% by putstatic_fast). Note that our solution allocates the object variables within the heap or a scoped region and the class variables within an immortal region.

In order to estimate the upper bound of write-barriers events, let s be the percentage of objects allocated by a task within scoped regions, and v the percentage of interspace assignments found in the total assignments made by the task. We assume that each object in the system has an equal probability of being referenced. Table II shows the maximum probability to execute write barriers when a task makes an assignment.

Note that in the software-based solution, checks to detect whether an application task attempts to violate the memory space of another application are

Table II. Upper Bounds on Write-Barrier Events

Event	Software	Hardware
Interspace checking	1	v + (1−v)s
Interregion checking	(1−v)	(1−v)s

Table III. Average Conditions Evaluated per Write-Barrier Event

Cost	Software	Hardware
Interspace checking	2	1
Interregion checking	1+n/2	0

always required; whereas the hardware-based solution saves violation-space checking whenever both objects X and Y belong to the same memory space and the Y object is not within a scoped region (i.e., the Y object is not within the heap or an immortal region, or within the common space).

4.2 Quantifying Cost

In order to quantify the *Cost* parameter, we consider the percentage of the execution time consumed by our write-barrier code in an assignment. For scoped regions, we must further consider the cost to have nested scoped levels, i.e., the cost to execute the nestedRegions(X,Y) function that is proportional to the number *n* of inner levels in the region-stack. We compute the execution time of write barriers as follows:

$$WriteBarrier\_Cost = Max\_Conditions * Condition\_Cost / Assignment\_Cost$$

Max_Conditions is the maximum number of evaluated conditions (see Table III), and *Condition_Cost* and*Assignemnt_Cost* are, respectively, the execution time to evaluate a condition, and the execution time of the original assignment code, which are constant for a given JVM implementation.

Our hardware-based solution then reduces both the number of events that require checks and the number of evaluated conditions required by each check. As consequence of both reductions, the average write-barrier cost per assignment has been reduced in more than 50% for interspace checks (i.e., from 2 to v + s − v*s; note that v< = 1 and s< = 1), and to 0 for interregion checks. More important, the write-barrier overhead does not depend on the number of nested regions entered by a task; note that determinism is an important property in real-time systems.

4.3 Memory Footprint

We have limited the number of memory spaces to 32. We consider that spaces are paged, the page size is 16 KB, and the maximum number of pages that a space can hold has been limited to 32. To limit the worst case for write-barrier execution time, we must also limit the number of scoped nested levels. Since we must take the *single-parent rule* of scoped regions [Bernadat et al. 1998] into account, 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.

```
configure_page_based
        priv_read_gc_config     // read the GC_CONFIG register
        spush 0x001F            // TRAIN_MASK=00000000000, CAR_MASK=11111
        seti 0x0888             // referenced-based WB_VECTOR=0000100010001000
        iand                    // and(GC_CONFIG, 0x001F0888)
        spush 0x001F            // TRAIN_MASK=00000000000, CAR_MASK=11111
        seti 0x0888             // referenced-based WB_VECTOR=0000100010001000
        ior                     // or(GC_CONFIG, 0x001F0888)
        priv_write_gc_config    // write the GC_CONFIG register
        priv_ret_form_trap      // exception returns
```

Fig. 12. Configuring page-based write barriers.

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. This increases the memory consumption in a word per object. We 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>, SPACE_ID <29:19>, REGION_ID <18:14>, SIZE <13:2>, MARK_BIT <1>, and STATIC_BIT <0>). Note that the maximum size of the object has been reduced from 16 MB to 2 KB; given the small average object size that the SPECJVM98 [SPEC JVM, 98] applications present (i.e., about 32 bytes) [Kin and Hsu 2000], we optimize for small objects. The TYPE field is also not required when using a nonaccurate GC.

We maintain a *region-structure* of two words for each memory region object in the system with the following format: REGION_TYPE <31:30>, REGION_ID <29:25>, OUTER_REGION_ID <24:20>, REFERENCE_COUNT <19:16>, SIZE <15:10>, SPACE_ID <9:5>, and PAGE <4:0>; where the REFERENCE_COUNT, 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 of the region in bytes, and the page supported in the region. The region structure increases the memory footprint by a maximum of 128 bytes. Note that these region structures form a *scope-tree* [Higuera-Toledo 2004], where the heap is the root and immortal regions are not included.

4.4 Configuring Write Barriers

Considering 32 spaces and a page size of 16 KB, we introduce (1) a routine to configure both page-based and reference-based write barriers (Figure 12), (2) a routine to enable page-based write barriers (Figure 13a) and a routine to disable page-based write barriers (Figure 13b), (3) a routine to enable memory

```
enable_gc_notify
        priv_read_psr
        spush   0x10000
        seti    0x0000
        iand                    // set  GCE
        priv_write_psr
        priv_ret_form_trap      // exc. returns
```

(a) Setting the GCE bit we enable write barriers

```
disable_gc_notify
        priv_read_psr
        spush   0xEFFF
        seti    0xFFFF
        iand                    // reset GCE
        priv_write_psr
        priv_ret_form_trap      // exc. returns
```

(b) Resetting the GCE bit we enable write barriers

Fig. 13. Enabling/disabling page-based write barriers.

```
enable_mr_notify
    priv_read_psr
    spush   0x0080
    seti    0x0000
    ior                         //set bit 23
    priv_write_psr
```

(a) Setting the MRE bit we disable write barriers

```
disable_mr_notify
    priv_read_psr
    spush   0xFF7F
    seti    0xFFFF
    iand                        //reset bit 23
    priv_write_psr
```

(b) Resetting the MRE bit we disable write barriers

Fig. 14. Enabling/disabling region-based write barriers.

region write barriers (Figure 14a) and a routine to disable memory region write barriers (Figure 14b).

5. CONCLUSIONS

Regarding our software-based solution, we found only two problems: the high overhead introduced by the dynamic checking for illegal assignment and that the introduced overhead must be bounded by limiting the nested scoped levels. Our solution to improve the performance of memory management resorts to hardware aid by exploiting existing hardware support for Java (i.e., the picoJava-II microprocessor). The performance of this solution has been highly improved. The introduced time overhead has been reduced to zero for allowed references within an immortal region or within a shared scoped region, and nearly zero for references within the local heap. By using specialized hardware, we can also considerably reduce the write-barrier cost for references within a nonshared scoped region; we omit write barriers in native code, which may be solved by forcing the native code to register their writes explicitly. This paper has presented a memory-management design solution for extending the RTSJ specification to execute several applications 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.

REFERENCES

BAKER, H. G. 1991. The treadmill: Real-time garbage collection without motion sickness. In *Proc. of Conference on Object and Oriented Programming, Systems Languages and Applications* OOPSLA.

BACK, G., TULLMANN, P., STOLLER, L., HESIENH, W. C., AND LEPREAU, J. 1998. Java operating systems: Design and implementation. *Technical rep. Department of Computer Science, University of Utah, http://www.cs.utah.edu/projects/flux* (Aug).

BERNADAT, P., LAMBRIGHT, D., AND TRAVOSTINO, F. 1998. 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*.

BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., HARDIN, D., AND MTURNBULL. (*The Real-Time for Java Expert Group*). Real-Time Specification for Java. RTJEG 2002. http://www.rtj.org.

CZAJKOWKI, G. 2000. 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 (Oct.).

HARDIN, D. S. 2001. Real-time objects on the bare metal. An efficient hardware realization of the Java virtual machine. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC).

HIGUERA-TOLEDANO, M. T., ISSARNY, V., BANATRE, M., CABILLIC, G., LESOT, J. P., AND PARAIN, F. 2004. Memory management for real-time Java: An efficient solution using hardware support. *Real-Time Systems Journal 26*, 1.

HIGUERA-TOLEDANO, M. T. 2004. Illegal References in a real-time Java concurrent environment. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC).

HIGUERA-TOLEDANO, M. T. 2003. Hardware-based solutions detecting illegal references in real-time Java". *In Proceedings of 15th Euromicro Conference on Real-Time Systems*.

ISSARNY, V., BANATRE, M., WEISS, F., WEIS, CABILLIC, G., COURDEC, P., HIGUER-TOLEDANO, M. T., AND PARAIN, F. 2000. Providing an embedded software environment for wireless PDAs. In *Proceegings of the Ninth ACM SIGOPS European Workshop—Beyond the PC: New Challenges for the Operating System* (Sept.).

JAVA COMMUNITY PROCESS. 2003. Application Isolation API Specification. *http://jcp.org/jsr/detail/121.jsp*.

JENSEN, D. W., GREVE, D. A., AND WILDING, M. M. Secure Reconfigurable Computing. Advanced Technology Center Advanced Technology Center. *http://www.klabs.org/richcontent/MAPLDCon99*, 1999.

KIN, S. AND HSU, Y. 2000. Memory system behaviour of Java programs: Methodology and analysis. In *Proceedings of the ACM Java Grande 2000 Conference.*

PALACZ, K., CZAJKOWSKI, G., DAINES, L., AND VITEK, J. 2002. Incommunicado: Efficient communication for isolates. In *Proceedings of the Conference on Object and Oriented Programming, Systems Languages and Applications* ACM OOPSLA (Nov.).

PETIT-BIANCO, A. AND TROMEY, T. 1998. Garbage collection for Java in embedded systems. In *Proceedings of IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Dec.).

SPRECJVMS98. 1998. Standard Performance Evaluation Council. SPEC JVM98 benchmarks. Technical report. *http://www.spec.org/osg/jvm98*.

SHOEBERL, M. 2005. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop,* RAW 2005, Denver, Colorado (Apr.).

SUN MICROSYSTEMS. 2000. picoJava-II Programmer's Reference Manual. Technical Report. Java Community Process (May). *http://java.sun.com*.

SUN MICROSYSTEMS. 2000. KVM Technical Specification. *http://java.sun.com* (May).

UHRIG, S., LIEMKE, C., PFEFFER, M., BECKER, J., BRINKSCHULTE, U., AND UNGERER, TH. 2002. Implementing real-time scheduling within a multithreaded Java microcontroller. In *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture and Compilation* MTEAC-6.

WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. 2002. A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France (Sept.).

WILSON, P. R. AND JOHNSTON, M. S. 1993. Real-time non-copying garbage collection. In *Proc. of Conference on Object and Oriented Programming, Systems Languages and Applications* ACM OOPSLA (*Workshop on Garbage Collection and Memory Management.*) (Sept.).

WONG, B., CZAJKOWSKI, G., AND DAYMES, L. 2003. Dynamically loaded classes as shared libraries: An approach to improving virtual machine scalability. In *Proceedings of the International Parallel and Distributed Processing Symposium* (IPDPS).

Received May 2004; revised July 2005; accepted January 2006