Dynamic Detection of Access Errors and Illegal References in RTSJ

M. Teresa Higuera-Toledano[1], Miguel A. de Miguel-Cabello[2], and Javier Resano-Ezcaray[1]

[1] Facultad Informática , Universidad Complutense de Madrid, Ciudad Universitaria, 28040 Madrid Spain
[2] Escuela Técnica Superiror de Telecomunicaciones, Universidad Politécnica de Madrid, Ciudad Universitaria, 28040 Madrid Spain
Email: mthiguer@dacya.ucm.es

Abstract

The memory model used in the Real-Time Specification for Java (RTSJ) imposes strict assignment rules to or from memory areas preventing the creation of dangling pointers, and thus maintaining the pointer safety of Java. This paper provides an implementation solution to ensure the checking of this rules before each assignment statement, where the check is performed dynamically by using write barriers.

Keywords: Java, Garbage-collector, Memory-regions, Real-time, Embedded, Write-barriers.

1. Introduction

Implicit garbage collection has always been recognized as a beneficial support from the standpoint of promoting the development of robust programs. However, this comes along with overhead regarding both execution time and memory consumption, which makes (implicit) garbage collection poorly suited for small-sized embedded real-time systems. However, there has been extensive research work in the area of making garbage collection compliant with real-time requirements. Proposed solutions may roughly be classified into two categories:

1. *Incremental garbage collection* (e.g., [1]) allows the application to execute while the Garbage Collector (GC) has been launched, a mechanism (i.e., read or write barriers) is used to coordinate the execution of the GC and of the application.
2. *Region-based memory allocation* (e.g., [3]) enables grouping related objects within a region. Commonly, Memory Regions (MR) are used explicitly in the program. This is an intermediate solution between explicit memory allocation/deallocation and the GC.

Application of these two strategies has been studied in the context of Java, which is in particular highlighted by the RTSJ [1]. This specification allows the implementation of real-time compliant GC without prescribing any specific solution. The `MemoryArea` abstract class supports the region paradigm in the RTSJ specification [11] 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 that may either offer temporal guarantees (i.e., supported by the `LTMemory` and LTPhysicalMemory classes) or not (i.e., supported by the `VTMemory` and `VTPhysicalMemory` classes) on the time taken to create objects; and (*iii*) the conventional heap, supported by the `HeapMemory` class. There is only one object instance of the `HeapMemory` and the `ImmortalMemory` classes in the system, which are resources shared among all threads in the system and whose reference is given by calling the `instance()` method. In contrast, for the `ImmortalPhysicalMemory`, and ScopedMemory classes, several instances can be created by the application.

Objects allocated within immortal MRs 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) GC of the JVM. A scoped region gets collected as a whole once it is no longer used. Since immortal and scoped MRs are not garbage collected, they may be exploited by hard real-time tasks, especially `VTMemory` objects, which guarantee allocation time proportional to the object size[1].

In this paper, we propose a GC strategy based on the tri-color algorithm complying with the RTSJ specification (Section 2). RTSJ imposes restricted assignments rules that keep longer-lived objects from referencing objects in life-limited memory regions (scoped regions). The proposed approach includes a new stack-based algorithm detecting illegal references when both objects, the one that makes the reference and the referenced one, are within scoped regions (Section 3). In Section 5, we implement a prototype within the KVM [13] by modifying the original GC and introducing MRs. Finally, a summary of our contribution concludes this paper (Section 5).

[1]The `ImmortalPhysicalMemory`, `VTPhysicalMemory`, and `LTPhysicalMemory` classes support regions with special memory attributes (e.g., *dma*, *shared*, *swaping*).

2. The Basic Collector Strategy

There are some important considerations when choosing a real-time GC strategy. Among them are space costs, barrier costs, and available compiler support. 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 an incremental non-copying GC based on the tri-color algorithm [2], which the basic algorithm is as follows: an object is colored *white* when not reached by the GC, *black* when reached, and *grey* when it has been reached, but its descendants may not be (i.e., they are white). Grey objects make a wavefront, separating the white (unreached) from the black (reached) objects, and the application must preserve the invariant that no black objects have a pointer to a white object, which is achieved by using *write barriers.*

For each thread, we maintain a stack of root pointers. We start the marking phase by coloring all objects referenced by root pointers grey. 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 tries 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 (see Figure 2).

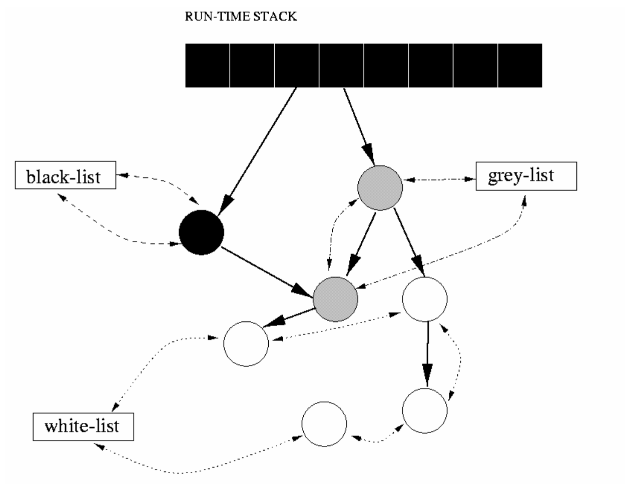


Figure 2. The GC strategy.

When all the descendants of a grey object are processed (i.e., the grey object has no white descendant), the grey object is turned black and moved from the grey-list to the black-list. The collection is completed when there are no more grey objects. During the sweeping phase, all the white objects can be recycled and all the black objects become white. In this process, objects that must execute the `finalize()` method are moved to the finalize-list, which are executed by a specialized thread such as in [10]. Black objects are marked white and moved to the white-list. Finally, for white objects that have finalized, their memory is freed. Then, a compacting phase can be added to move objects into a continuous block into the heap.

2.1. Write Barrier Strategy

The code checking a reference violating the tri-color invariant (i.e., from a black object to a white one) must be executed when updating an object reference (i.e., when executing the `putfield`, `putstatic`, `aastore`, `aputfield_quick`, `aputstatic_quick`, or `aastore_quick` bytecodes). In order to maintain the tri-color invariant, we introduce in the interpretation of these bytecodes, the write barrier pseudo-code shown in Figure 3, where we denote as X the object that makes the reference, and as Y the referenced object, the `color()` function gives the color of the object parameter, and the `greyObject(Y)` procedure unlinks the Y object from the white-list linking it to the grey-list [5].

```
WriteBarrierGC:
    if ((color(X) = black) and (color(Y) = white)) greyObject(Y);
end_writeBarrierGC;
```

Figure 3. Write barrier code for the tri-color invariant.

2.2. Dealing with Fragmentation

In general, memory fragmentation is not a severe problem. Additionally, if the application only allocates objects with small size, acceptable worst-case bounds on fragmentation can be given. For objects with large size, strategies such as fragmenting the object into smaller-size chunks may be used. Another strategy relies on occasionally running a compacting GC, which implies some degradation of real-time guarantees.

Given the small average object size that Java applications present (i.e., between 25 and 40 bytes [8]), it appears imperative to keep the number of header words to a minimum. Since objects outside the heap (i.e., objects allocated within immortal or scoped MRs) are not moved, we improve the performance by avoiding object handles and hence use direct reference objects. Note that this strategy is always possible; we can avoid handles for objects outside the heap, even if we add a compactation phase to our GC, requiring handles for objects within the heap. When eliminating handles we improve also memory consumption in a word per object, which means approximately 11% of the total required object space, given that the average size of Java objects is 32Bytes.

Hence, we found more interesting to not use a compacting phase, and to avoid handles.

2.3. The GC and MR

Since objects allocated within regions may contain references to objects within the heap, the GC must take into account these external references, adding them to its reachability graph. To detect when an object outside the heap references an object within the heap, we introduce a fourth color (e.g., *red*) meaning that the object is allocated outside the heap (See Figure 5). Hence, we introduce a new invariant:

Definition- *Fourth-color invariant*: *there are no red objects within the heap, and all outside the heap are red.*

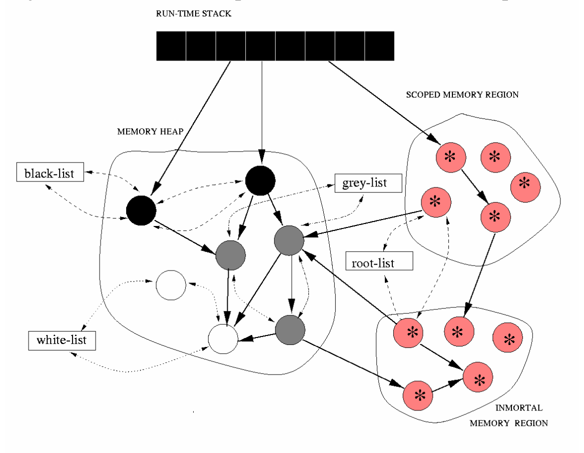


Figure 5. Objects outside the heap are allocated red.

The fourth color allows us to detect when the X object must be added to the root-set of the collector, where the root-list is updated (i.e., by using write barriers). A reference from a red object (X) to another object (Y) allocated in the heap (i.e., white, black, or grey) causes the addition of the X object to the root-set of the collector, which is achieved by using write barriers, as shows the code of Figure 6, where the updateRootSet(X, Y) procedure links the X object to the root-list blackening it and greying the Y object if it is white. 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 a scoped MR ends, all objects within the region having references to the objects within the heap are removed from the root-list of the collector.

```
writeBarrierGC:
    if ((color(X) = red) and  (color(Y) <> red)) updateRootSet(X, Y)
    else  if ((color(X) = black) and (color(Y) = white)) greyObject(Y);
```

Figure 6. Allocating grey object outside the heap.

RTSJ makes distinction between three main kinds of tasks: *(i) low-priority* that are tolerant with the GC, *(ii) high-priority* that cannot tolerate unbounded preemption latencies, and *(iii) critical* that cannot tolerate preemption latencies. Low-priority tasks, or threads, are instances of the `Thread` class, high-priority tasks are instances of the `RealtimeThread` class, which extend the `Thread` class to support real-time tasks, and critical tasks are instances of the `NoHeapRealtimeThread` class, which extend the `RealtimeThread` class to avoid critical task have delays because the GC[2].

2.4. Dealing with Critical Tasks

Whereas high-priority tasks require a real-time GC, critical tasks must not be affected by the GC, and as a consequence cannot access any object within the heap [11]. A reference of a critical task to an object allocated in the heap causes the `MemoryAccessError()` exception, which can be achieved by using *read barriers*. Note that read barriers occur upon all object accesses, which means upon executing both types of bytecodes: *(i)* Those causing a *load reference* (i.e., `getfield`, `getstatic`, `agetfield_quick`, `agetstatic_quick`, or `aaload` bytecodes). *(ii)* Those causing a *store reference* (i.e., those causing write barriers: `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, or `aastore_quick` bytecodes).For bytecodes causing a load reference, we introduce the read barrier code given in Figure 7, where the `type()` function returns `thread`, `task`, and `critical` depending on the type of the parameter task, *t* is the active task, and the code tagged `memoryAccessError:` throws the `MemoryAccessError()` exception. For bytecodes causing a store reference (i.e., those executing write barriers), we modify the `writeBarrier` pseudo-code to integrate the read barriers (see Figure 8).

```
readBarrier:
    if   ((type(t)   =   critical)   and   (color(X)   <>     red)     goto
memoryAccessError::
```

Figure 7. Detecting accesses of critical tasks into the heap.

```
writeBarrier:
   if ((type(t) = critical) and (color(Y) <> red))  goto
memoryAccessError:;
    if ((color(X) = red) and (color(Y) <>red)) updateRootSet(X, Y)
```

Figure 8. Detecting illegal accesses from critical tasks.

Note that read barriers are not strictly necessary because read operations do not change the color of the object [4]. Here, we apply the same optimization as for the incremental *Treadmill* GC [1]which is to use write barriers instead of read barriers, and the restriction on critical tasks

[2] In RTSJ, the NoHeapRealtimeThread class specializes RealtimeThread, that extends java.lang.Thread for real-time.

can be reduced to write barriers checks since reads does not interfere with the GC. In this way, we check that a critical task never modify the graph of the collector instead to check that a critical task never access an object within the heap. Since accesses to objects within the heap does not requires synchronization between the GC and the application (i.e., the synchronization is made only when modifying the graph of the collector), a critical task can preempt immediately the GC, even if accesses to objects within the heap are allowed. Then, we change the `MemoryAccessError()` exception which raises when a critical task attempts to access an object X within the heap by the `IllegalAssignmentError()` exception which raises when a critical task attempts to assign an object Y which belongs to the heap (see Figure 9).

```
writeBarrier:
    if ((type(t) = critical) and  (color(Y) <>  red)) goto
illegalAssignment:;
    if ((color(X) = red) and (color(Y) <> red)) updateRootSet(X, Y)
```

Figure 9. Detecting illegal assignments from critical tasks.

3. Scoped Regions

Several research have examined the possibility of replacing the Java garbage collection by an adequate stack-allocation scheme, which is more predictable [11]. Stack-allocation is desirable because execution time properties are easier to capture than heap allocation. To support scoped memory regions, we propose a mechanism based on a reference-counter collector and a scoped region-stack based algorithm. Then, every scoped region is associated with a reference counter that keeps track of the use of the region by tasks. And every task is associated with a stack that keeps track of the scoped region that can be accessed by the task. In this section, we describe the main principles of the proposed algorithms.

3.1. Checking Illegal Assignments

The lifetime of objects allocated in scoped regions is governed by the control flow. Strict assignment rules placed on assignments to or from MRs prevent the creation of dangling pointers (see Table 1 [11]).

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Same, outer, or shared

Table 1. Assignment rules in RTSJ.

An implementation must ensure that the following conditions are checked before the assignment is executed: *(i)* objects within the heap or within the immortal region cannot reference objects within a scoped region and *(ii)* objects in a scoped region cannot reference objects within another scoped region that is non-outer. Then, a code checking *illegal assignments* and throwing the `IllegalAssignmentError()` exception when detecting an attempt of illegal assignment must be added when updating an object reference (see Figure 10).

```
writeBarrierMR:
    if        ((region(X)<>scoped)and(region(Y)=scoped))        goto
illegalAssignment:;
    if ((region(X) = scoped)and(region(Y) = scoped))  nestedRegions(X,
```

Figure 10. Write barrier code detecting illegal assignment.

The algorithm to check illegal references from a scoped region to another scoped one (i.e., the `nestedRegions(X, Y)` function [7]) is following described, before which we give an algorithm supporting the *region-stack* used when checking nested scoped regions.

3.2. Region-Stack Algorithm

In order to detect illegal assignments to scoped regions, every thread has associated a region-stack containing all scoped MRs which the thread can hold. The MR at the top of the stack is the active region for the task whereas the MR at the bottom of the stack is the outermost scoped region for the task. The default active region is the heap. When the task does no use any scoped region, the region-stack is empty and the active region is the heap or an immortal MR. Both, the active region and the region-stack associated with the task change when executing the enter() or `executeInArea()` methods.

Region-Stack Algorithm:

- When creating a new scoped region (i.e., a new `LTMemory`, `VTMemory`, `LTPhysicalMemory`, or `VTPhysicalMemory` object), a new region-stack is associated with the new region. The associated region stack is composed of the region-stack of the outer scoped region, which can be empty (e.g., when the active region is the heap), and the identifier of the new scoped region which is added on the top of the stack.
- When creating a new task (i.e., a `RealtimeThread` or `NoHeapRealtimeThread` object), a memory region is associated with it[3]. Then, if the active region is a

[3] The associated memory region is specified through the RealtimeThread and NoHeapRealtimeThread constructors.

scoped one, the region-stack of the memory region is associated with the task. Both the active region and the active region-stack are considered part of the part of the task's state, which must be saved/restored at context change time. Note that by only saving the old active region value, we can obtain both the active and the region-stack pointer.

- When a task enters a region through the `enter()` (`executeInArea()`) method, the active region is changed to the entered region, and as consequence the region-stack associated with the task. Then, the old active region must be saved, before to change it. Both, the old active region and the region-stack are restored when returning from the `enter()` (`executeInArea()`) method. Note that when this happens the reference counter of the scoped region which is at the top of the stack must be decremented. We detail hereafter the main steps of the algorithm.

3.4. Checking Nested Regions

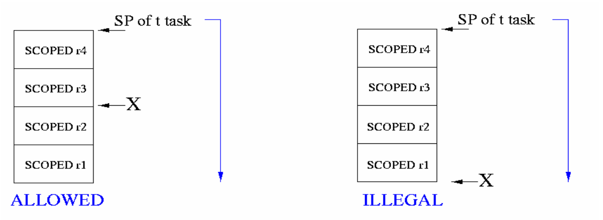
The basic idea to detect illegal assignments is to take actions upon those instructions that cause one object to reference another [4] (i.e., we must introduce write barriers).

- The `putfield` (`aputfield_quick`) bytecode causes a reference from an object (X) to another one (Y), and the `aastore` (`aastore_quick`) bytecode stores a reference (Y) into an array of references (X). Then, the scope of X must be inner than the scope of Y.
- The `putstatic` (`aputstatic_quick`) bytecode causes a reference from the outermost region (i.e., the heap) to an object Y.
- The region to which an object belong must be specified in the header of the object. 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 the active region. Following we describe the `nestedRegions(X, Y)` function.

Nested MR Algorithm: nestedRegions(X, Y)

Checking nested regions requires two steps. In a first step, the region-stack of the active task is explored, from the top to the bottom, to find the MR to which the X object belongs (see Figure 11.a). If it is not found (see Figure 11.b), this is notified by throwing a `MemoryAccessError()` exception[4].

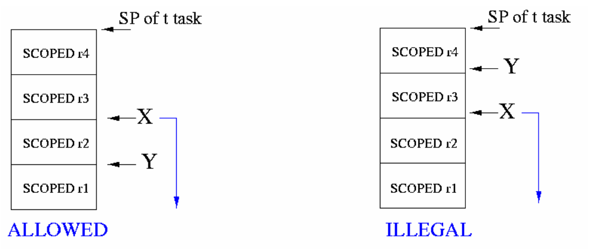
[4] This exception is thrown upon any attempt to refer to an object in an inaccessible MemoryArea.



a. The region of X is found. b. The region of X is not found.

Figure 11. First exploration of the region-stack.

In a second step, the region-stack is again explored, but this time we take the MR found in the previous step as the top of the stack (see Figure 12.a). Then, we start the search from the region to which the X object belongs, and the objective is to find the MR to which the Y object belongs (i.e., the region to which the object Y belongs must be outer to the region to which the object X belongs). If the scoped region of Y is not found in the new region-stack (see Figure 12.b), (i.e., the heap that is the outest region and hence at the bottom of the stack is reached), this is notified by throwing a `IllegalAssignmentError()` exception. If it is found, the `nestedRegions(X, Y)` returns `true`.



a. Y is outer to X . b. Y is inner to X.

Figure 12. Second exploration of the region-stack.

Since the `nestedRegions(X, Y)` function, which executes the region stack algorithm detects illegal assignments from objects within non-scoped regions to objects within scoped regions raising the `IllegalAssignmentError()`, we use the code given in Figure 13 as a final solution.

```
writeBarrier:
      if (region(Y) = scoped)
            if (region(X) =  scoped)  nestedRegions(X,Y)
            else goto illegalAssignment:;
      if ((type(t) = critical) and (region(Y) = heap))  goto
illegalAssignment:;
      if ((color(X) = red) and (color(Y) <> red))  updateRootSet(X, Y)
```

Figure 13. Write barrier code for both the GC and MRs.

3.5. Scoped Region Collection

A safe region implementation requires that a scoped MR gets deleted only if there is no *external* reference to it.

The problem presented by nested regions can be solved by using a reference-counter for each region, and a simple reference-counting GC collects scoped MRs when their counter reaches zero [1]. Note that by collecting regions, problems associated with reference counting collectors are solved: the space and time to maintain a reference counter per scoped MR is minimal, and there is no cyclic MR reference. The reference-counter is increased when associating the region to a task (i.e., when a thread enters a new scoped region through the `enter()` (`executeInArea()`) method or when creating a real-time thread with a scoped region through the `RealtimeThread` or `NoHeapRealtimeThread` constructors), or when opening an inner scoped region.

It is decreased when a task leaves the region (i.e., when returning from the `enter()` (`executeInArea()`) method or when the task which uses the scoped region exits), or when an inner scope ends. Note that for the heap and immortal MRs, there is no need to maintain a reference counter because these regions exist outside the scope of the application, that creates the objects. Recall also, that references from objects within the heap, an immortal MR, or a scoped MR, to objects within the heap or immortal memory are allowed.

Scoped Region Collector Algorithm:

- When creating a new scoped region, its reference-counter is initialized to zero.
- When assigning a scoped region to a variable or to a field object:
    1. If the variable or the field object references a scoped region, the reference-counter of the scoped region that lost the reference is decremented.
    2. The reference-counter of the scoped region is incremented.
- When starting the execution of a task using a scoped region or when a task enters a scoped region, the reference-counter of the region is incremented.
- When a task exit or when the enter() (`executeInArea()`) method returns, if the exited region is a scoped one, the reference-counter of the region is decremented.
- When collecting a scope region because its reference counter reaches zero:
    1. The root-list of the GC is update to remove all the objects in the region that are external roots for the GC.
    2. All the objects in the region are moved to the finalize-list, where their `finalize()` method is executed.
    3. If the scoped MR belongs to another scoped MR, the reference counter of the outer region is decremented.

Then, when removing a region, it is sure that there is no object dependent on an older scoped region.

4. Experiment

We have modified the KVM [13] garbage collector[5] making it incremental byusing the tri-color algorithm. We have implemented the `MyIncrementalGC` class within the KVM by modifying some files[6]. This class supports the method related with parameters characterizing the collector behavior (e.g., `getPreemtionLatency()`, `getMinimumReclamationRate()`, `getOverhead()`, and `getWriteBarrierOverhead()` methods). We have only implemented three types of memory regions: *(i)* the heap that is collected by an incremental GC, *(ii)* immortal that are never collected and can not be nested, and *(iii)* scoped that have limited live-time and can be nested. These regions are supported by the `HeapMemory`, the `ImmortalMemory`, and the `ScopedMemory` classes. Unlike RTSJ, in our prototype the `ScopedMemory` class is a non-abstract class, and the `MemoryArea` abstract class has not been implemented[7]. The `getWriteBarrierOverhead()` method has been implemented for the four classes, to give the percentage of the execution cost introduced by the write barrier code for the original execution cost of each assignment.

We have limited to 256 the number of regions and the number of scoped nexted levels to 8, which allows us to support the region stack of each task in 4 words. The original header format of KVM objects (i.e., SIZE <31:8>, TYPE <7:2>, MARK_BIT <1>, and STATIC_BIT <0>) has been modify to support the color and region of the object (i.e., SIZE <31:15>, REGION <14:8>, TYPE <7:2>, COLOR <1:0>). Note that size of the object header has not been incremented, instead the maximum object size has been reduced from 32 Mbytes to 64 KBytes. The MARK_BIT that is used by the original mark-and-sweep collector of the KVM to mark the object is not longer used because objects are market by color, also the STATIC_BIT is not used because it came from an old collector based on the copying algorithm that have been changed in order to make the KVM suitable to small devices. The maximum heap size supported by the KVM is 32 Mbytes, as in the original version.

[5] Version 1.0.1

[6] We have modified the `garbage.c` file to implement the collector algorithm and the `interpreter.c` file to implement the write barriers, as well as the `native.h` and the `nativeCore.c` files, which support the interface for the native methods.

[7] This due to the limitations of heritage in the KVM.

We use an artificial collector benchmark which is an adaptation made by Hans Boehm from the John Ellis and Pete Kovac benchmark[8]. This benchmark executes $262*10^6$ bytecodes and allocates 408 MBytes. Then, the allocation rate is about 1.6 Bytes per executed bytecode. The maximum latency to preempt the incremental collector has been measured as 1 micro-second. The number of garbage collection pas, the seconds spent in garbage collection, the seconds spent in execute de application, and the percentage overhead introduced by our collector is given in Table 2.

Table 2. Assignment rules in RTSJ.

4.1 Write Barrier Overhead

In RTSJ, the `getWriteBarrierOverhead()` method of the `IncrementalGarbageCollectorclass` gives the write barrier cost per assignment, i.e., writeBarrierCost/assignmentCost where the writeBarrierCost is the execution time of the introduced write barriers, and the assignmentCost is the execution time of an object assignment. Thus, we compute writeBarrierCost for an incremental GC, as the cost to detect when to take actions preserving the tri-color invariant, i.e., the execution time taken to detect when to execute the `greyObject(Y)` function:

```
if ((color(X) = black) and (color(Y) = white)) greyObject(Y);
```

Note that the execution time taken by the `greyObject(Y)` function is considered as part of the GC overhead rather than as part of the write barrier overhead. The `GarbageCollector` abstract class of RTSJ does not support the `getWriteBarrierOverhead()` method12. Since the heap coexists with other MRs, we consider that this method must also be implemented for all collectors to give the overhead caused by detecting illegal assignments of critical task to objects within the heap. For mark-and-sweep collectors, this method further gives the overhead caused by the write barriers introduced to detect when to update the collector's root-set:

```
If ((color(X) = red) and (color(Y)<>red))  updateRootSet(X, Y);
```

Minimizing the Write Barrier Overhead.
The most common approach to implement write barriers is by inline code, consisting in generating the instructions executing write barrier events for every store operation. This solution requires compiler cooperation (e.g., JIT), and presents a serious drawback because it nearly doubles the application's size. Regarding systems with limited memory such as PDAs, this code expansion overhead is considered prohibitive. Alternatively, we can instrument the bytecode interpreter, avoiding space problems, but this still requires a complementary solution to handle native code. A solution minimizing the write barrier overhead consists in improving the write barrier performance by using hardware support such as the picoJava-II microprocessor [15], which allows performing write barrier checks in parallel with the store operation. This alternative solution has been the subject of [8].

The number of executed bytecodes performing write barrier test is $15*10^6$ (i.e., `aastore:` $1*10^6$, `putfield:` $6*10^6$, `putfield_fast:` $7*10^6$, `putstatic:` $19*10^6$, and

Memory Heap	GC pass	Collecting Time	Execution Time	% Overhead
8 MB	51	13.54	72.87	18.85
16 MB	27	13.17	72.72	18.11
24 MB	17	12.80	71.99	17.80
32 MB	13	11.82	70.50	16.50

`putstatic_fast:` 0) for a total of $262*10^6$ executed bytecodes. This means that 5% of executed bytecodes perform a write barrier test, as already obtained in [6], and the overhead introduced by the software write barrier test in each assignment is:

- 45% to maintain the root-set.
- 31% to preserve the tri-color invariant.
- 31% to detect illegal references.
- 16% to check a nested scoped level.

Then, the RTSJ-instrumented KVM runs slowdown as minimum 5,35% and as maximum 11,75% than the original KVM (i.e., 5,35,+0,80*n; where n is the maximum number of allowed nexted levels) .

5. Conclusions

The current RTSJ specification imposes restricted assignments rules that keep longer-lived objects from referencing object in scoped memory, which are possibly shorter live. This requires run-time checks for each assignment, which introduces a high overhead. In the RTSJ model, the way to offer real-time guarantees is by turning off the GC during the execution of critical tasks, which only allocates objects in memory regions and cannot reference objects within the heap. Some real-time tasks can allocate and reference objects within the heap, whereas others (critical) are no allowed to allocate nor reference objects within the heap. This requires run-time checks for all the object accessed by all the application tasks, which introduces a high overhead.

We have proposed a solution to the realization of the abstract memory model introduced by the RTSJ specification. In particular, garbage collection in the heap complies with real-time constraints by using write barriers to maintain both the root-set and the tri-color invariant. In our solution, the detection of illegal assignments related with memory regions and access errors related with critical

[8]http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html

tasks, is made dynamically by introducing a write barrier mechanism based on a region-stack associated to the active task.

Related Work

The JVM must check for illegal references and throw an exception if they occur. In order to do that, we introduce extra code that must be executed when updating an object reference (i.e., write barriers), which introduces high overhead. A similar approach is given in [], which uses also a stack-based memory management that operates dynamically. This solution proposes a *contami nated GC* based on the idea each object in the heap is alive due to references that begin in the runtime stack. But this solution collects memory within the heap, and does not treat another memory region. A safe implementation requires that a region can be deleted only if there is no *external* reference to it. The Tofte-Talpin calculus [tt] uses a lexically scoped expression to delimit the lifetime of a region. Memory for the region is allocated when the control enters the scope of the region constructor, and is de-allocated when the control leaves the scope. This mechanism is implemented by a stack of regions where regions are ordered by lifetimes. The allocation and de-allocation of regions is determined at compile time by a type-based analysis, consisting to annotate in the source program every expression creating a value with a region variable []. An intermediate la nguage allows checking the safety of arbitrarily ordered regions, where region allocation and de-allocation are explicit.

As in [tt], our solution is based on a stack of scoped regions, where regions are ordered by life-times. But given that in RTSJ, a region can be shred by several threads, this solution requires more complex mechanisms because the region will remain active until the last thread has exited. Then, the de-allocation of regions can-not be determined at compile time. As in [], this problem has been resolved in RTSJ by using a *reference counter* for each region. The counter is incremented (decremented) when creating (collecting) an inner scoped region, and in our solution also when the region is associated to (de-associated from) a task.

Since in RTSJ the collector coexist with memory regions, objects within the heap having references from objects outside the heap must be considered as roots by tracing-based collectors (i.e., mark-and-sweep, incremental, or generational). In order to detect new roots of the collector, we introduce a color indicating whether the object is outside the heap, and use write barriers [RT-systems journal]. In order to characterize the write barrier overhead introduced by both critical tasks and the collectors roots, we add the `getWritebarrierOverhead()` method to the RTSJ `GarbageCollector` abstract class. For subclasses of this abstract class supporting write barrier-based collectors (i.e., incremental or generational), the `getWritebarrierOverhead()` method gives the overhead to maintain the tri-color invariant or inter-generational pointers. In order to characterize the write barrier cost to detect inter-region assignements from objects within non-scoped regions to objects within scoped scoped regions, we add the `getWritebarrierOverhead()` method to the `MemoryArea` abstract class. This method must be rewritten on the Scoped Memory abstract class (subclass of `MemoryArea`) to give the cost to detect assignments from objects within scoped regions to objects within a non-outer scoped region. Several ways to improve the performance write barriers has been proposed in [] and []. The performance of the software-based solution presented in this paper have improved in [] by (a) using existing hardware support, and (b) modifying existing hardware. The performance of these three solutions, the software-based and the two hardware based solutions, have been compared in []. At different that [], [], and [], which address the performance of write barriers and ways to improve it, this paper describe with detailed the software-based solution more precisely .

References

[1] H.G. Baker. "The Treadmill: Real-Time Garbage Collection without Motion Sickness" *.In Proc. of the Workshop on Garbage Collection in Object-Oriented Systems.* OOPSLA'91. ACM 1991.Also appears as SIGPLAN Notices Vol. 27, no. 3, pages 66-70, March 1992.

[2] G. Bollella and J. Gosling."The Real-Time Specification for Java". IEEE Computer, June 2000.

[3] E.W. Dijstra, L. Lamport, A.J. Martín, C.S. Scholtenand, and E.F.M. Steffens. "On-the-fly Garbage Collection: An Exercise in Cooperation". Communications of the ACM, 21(11):965-975, November 1978.

[4] D. Gay and A. Aiken. "Memory Management with Explicit Regions". *In Proc. of the Conference of Programming Language Design and Implementation (PLDI).* ACM SIGPLAN 199

[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. Kluber Academic Publishers, to be published.

[6] M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Region-based Memory Management for Real-time Java". *In Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC). IEEE 2001.

[7] M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Analyzing the Performance of

Memory Management in RTSJ”. *In Proc. of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC). IEEE 2002.

[8] M.T. Higuera. "Memory Management Solutions for Real-time Java". PHD Thesis. INRIA-Rocquencourt, March 2002.

[9] J.S. Kim and Y. Hsu. "Memory System Behavior of Java Programs: Methodology and Analysis". *In Proc. of the ACM Java Grande 2000 Conference*.

[10] A. Miyoshi, H. Tokuda, and T. Kitayama. "Implementation and Evaluation of Real-Time Java Threads". *In Proc. of the Real-Time Systems Symposium.* IEEE December 1997.

[11] A. Petit-Bianco and T. Tromey. "Garbage Collection for Java in Embedded Systems". *In Proc. of IEEE Workshop on Programming Languages for Real-Time Industrial Applications.* December 1998.

[12] The Real-Time for Java Expert Group. "Real-Time Specification for Java". RTJEG 2002 . http://www.rtj.org

[13] A. Reid, J. McCorquodale and J. Baker. "The Need for Predictable Garbage Collection". *In Proc. of Workshop on Compiler Support for System Software,* WCSSS'99. ACM SIGPLAN 1999. http://www.cs.utah.edu/projects/flux.

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