Towards an Understanding of the Behavior of the Single Parent Rule in the RTSJ Scoped Memory Model*

M. Teresa Higuera-Toledano

Facultad Informática, Universidad Complutense de Madrid, Ciudad Universitaria, 28040 Madrid Saín

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. An implementation solution to ensure the checking of these rules before each assignment statement consists of performing it dynamically by using write barriers. This solution adversely affects both the performance and predictability of the RTSJ application. In this paper we present an efficient algorithm for managing scoped regions which requires some modifications in the current RTSJ specification.

Keywords: Real-time Java, Scoped-regions, Garbage collection, Write-barriers.

1. Introduction

From a real-time perspective, the Garbage Collector (GC) introduces unpredictable pauses that are not tolerated by real-time tasks. Real-time collectors eliminate this problem but introduce a high overhead. An intermediate approach is to use memory regions within which allocation and de-allocation are customized and also space locality is improved. Application of these two implicit strategies has been studied in the context of Java; they are combined in the *Real-time Specification for Java* (RTSJ) [16], which introduces the concept of scoped memory to Java. RTSJ extends the Java memory model by providing several kinds of memory regions, among them the *garbage-collected heap*. RTSJ memory regions have different properties in term of both the object lifetimes and the object allocation/de-allocation timing guarantees. Particularly, *immortal memory* regions are never garbage collected, and *scoped memory* regions are collected when there is not a thread using the area. The garbage collector within the heap must scan all objects allocated within immortal or scoped memory regions for references to any object within the heap in order to preserve the integrity of the heap.

Because scoped regions can be reclaimed at any time, objects within a region with a longer lifetime are not allowed to create a reference to an object within another region with a potentially shorter lifetime. An RTSJ implementation must enforce these scope checks before executing an assignment. A possible solution is to perform these checks dynamically, each time a reference is stored in the memory (i.e., by using *write barriers*) [9].

This paper focuses on the data structure and the algorithms used to implement an alternative RTSJ memory model based on both the first [2] [15] and current edition of RTSJ [16], which allow us to enforce the safety assignment rules in a more efficient way. The main contribution of this paper is that of showing how all the necessary run time checks can be performed in constant time by simplifying the scoped memory hierarchy. This allows us to avoid the single-parent rule checks and to use a range-based encoding to implement dynamic scoped checks.

In this paper, we first present an in depth description of the RTSJ memory model regarding its current implementation techniques (Section 2). Then, we present our solution to improve the RTSJ suggested memory model implementation, showing how our alternative model can be implemented efficiently by using simple data structures and algorithms (Section 3). We include the maintaining of a display-based technique which uses to check illegal references makes the management of the RTSJ scoped memory areas time-predictable, presenting some results of performance analysis (Section 4). We provide an outline of the state of the art and related work (Section 5). Finally a summary of our contribution together with an overview of our ongoing work towards offering an overall memory management solution for real-time Java systems conclude this paper (Section 6).

* Founded by the Ministerio de Ciencia y Tecnología of Spain (CICYT); Grant Number TIC2003-01321.

2 The RTSJ memory model

The `MemoryArea` abstract class supports the region paradigm in RTSJ 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. In the following, we study how these memory regions are used by a real-time application.

2.1 The memory model behavior

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 the `instance()` method. In contrast, for the `ImmortalPhysicalMemory` and `ScopedMemory` classes, several instances can be created by the application. An application can allocate memory into the system heap, the immortal system memory region, several scoped memory regions, and several immortal regions associated with physical characteristics.

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 dangling pointers (see Table 1).

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Same or outer
Local Variable	Yes	Yes	Same or outer

Table 1: Assignment rules in RTSJ.

Scoped areas can be nested and each scope can have multiple sub-scopes, in this case the scoped memory hierarchy forms a *tree*. Consider two scoped memory regions, A and B, where the A scoped region is parent of the B region. In such a case, a reference to the A scoped region can be referenced from a field of an object allocated in B. But a reference from a field of an object within A to another object allocated in B raises the `IllegalAssignment()` exception. When a thread *enters* a scoped region, all subsequent object allocations come from the entered scoped region. When the thread *exits* the scoped region, and there are no more active threads within the scoped region, the entire memory assigned to the region can be reclaimed along with all objects allocated within it.

2.2 The task model behavior

Also, RTSJ makes a 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 tasks that have delays because of the GC[1] [16]. Since immortal and scoped regions are not garbage collected, they may be exploited by critical tasks, especially `LTMemory` and `LTPhysicalMemory` objects, which guarantee allocation time proportional to the object size[2].

Several related threads, possibly real-time, can share a memory region, and the region must be active until at least the last thread has exited. The way that threads access objects within memory regions in the current RTSJ edition is governed by the following rules:

1. A traditional thread (i.e., a `Thread` object) can allocate memory only within the traditional heap.

2. High-priority tasks (i.e., `RealtimeThread` objects) may allocate memory within the heap, or within a

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

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

memory region other than the heap by making that region the current allocation context.

3. Critical tasks (i.e., `NoHeapRealtimeThread` objects) must allocate memory from a memory region other than the heap by making that area the current allocation context.

4. A new allocation context is entered by calling the `MemoryArea.enter()` method, the `MemoryArea.executeInArea()` method, or by starting a real-time thread (i.e., a task or an event handler) whose constructor was given a reference to an instance of the `MemoryArea` abstract class. Once a region is entered, all subsequent uses of the `new` keyword, within the program logic, will allocate objects from the memory context associated to the entered region. When the region is exited, all subsequent uses of the `new` operation will allocate memory from the region associated with the enclosing scope.

5. Each real-time thread is associated with a scoped stack containing all the regions that the thread has entered but not yet exited.

   In the former RTSJ edition [15], this rule does not appear.

2.3 Scoped region behavior

For the scoped region model behavior, the current edition of the RTSJ specification has the following rules:

1. The structure of enclosing scopes is accessible through a set of methods on the `RealtimeThread` class, which allows outer scopes to be accessed like an array (e.g., the `getOuterScope()` method).

   In the former RTSJ edition [15], this rule has been formulated, as follows: *The `getOuterScope()` method of the **`ScopedMemory`** (instead `RealtimeThread`) class allows us to know, for the current thread, the memory region prior to entering the active region (i.e., the ancestor of the current active region).*

2. Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count of the number of threads in which it is being used.

   In the former RTSJ edition [15], this rule has been formulated, as follows: *Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count of the number of **scopes** (instead threads) in which it is being used.*

4. When the reference count for an instance of the class `ScopedMemory` is decreased from one to zero, all objects within that area are considered unreachable and are candidates for reclamation. The *finalizers* for each object in the memory associated with an instance of `ScopedMemory` are executed to completion, before any statement in any thread attempts to access the memory area again.

5. The *parent* of a scoped region is the region in which the object representing the scoped region is allocated.

   In the former RTSJ edition [15], this rule does not appear.

6. The *single parent rule* requires that a scoped region has exactly zero or one parent.

   In the former RTSJ edition [15], this rule does not appear.

7. Scoped regions that are made current by entering them or passing them as the initial memory area for a new task must satisfy the single parent rule.

3 Our RTSJ suggested scoped memory model

In the current RTSJ, when a task or an event handler tries to enter a scoped region S, we must check if the corresponding thread has entered every ancestor of the region S in the scoped region tree. Then, the safety of scoped regions requires both checking the set of rules imposed on their entrance and checking the aforementioned assignment rules. Both tests require algorithms, the cost of which is linear or polynomial in the number of memory regions that the task can hold. Also, in practice we have not found real-time application scenarios that would require more than a handful of scoped regions. We suppose that the most common RTSJ uses a scoped area to repeatedly perform the same computation in a periodic task. Then, to optimize the RTSJ memory subsystem, we suggest simplifying data structures and algorithms. In order to do that, we propose to change the RTSJ suggested implementation[3] of the parentage relation for scoped regions [16].

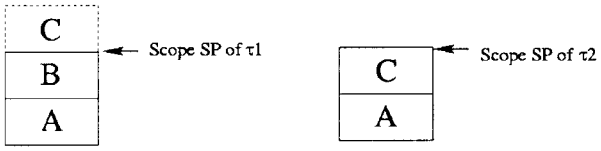
[3] Note that we do not propose to change the scoped regions parentage relation, but its suggested implementation.

3.1 The RTSJ single parent rule

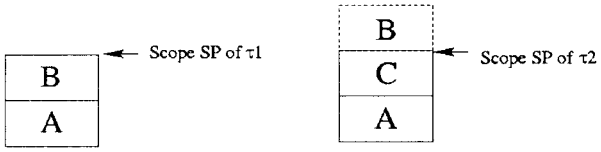
Therefore, the single parent rule guarantees that a parent scope will have a lifetime that is not shorter than of any of its child scopes, which makes safe references from objects in a given scope to objects in an ancestor scope, and forces each scoped region to be almost once in the tree containing all region stacks associated with the tasks that have entered the regions supported by the tree. The single-parent rule also enforces every task that uses a region to have exactly the same scoped region parentage.

The implementation of the single-parent rule as suggests the current RTSJ edition [16] makes the behavior of the application non-deterministic. In the guidelines given to implement the algorithms affecting the scope stack (e.g., the `enter()` method), the single parent rule guarantees that once a thread has entered a set of scoped regions in a given order, any other thread is enforced to enter the set of regions in the same order.

Consider three scoped regions: A, B, and C, and two task $\tau 1$ and $\tau 2$. Where task $\tau 1$ wants enter the regions as follows: A, B, and C, whereas $\tau 2$ wants to enter the regions in the following order: A, C and B. Let us suppose that task $\tau 1$ has entered regions A and B, and task $\tau 2$ has entered regions A and C. If task $\tau 1$ tries to enter the region C (see Figure 1.a) or task $\tau 2$ tries to enter the region B (see Figure 1.b), the single parent rule is violated and as consequence the `ScopedCycleException()` throws.



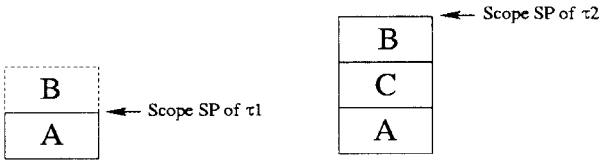
a. τ1 violates the single parent rule.



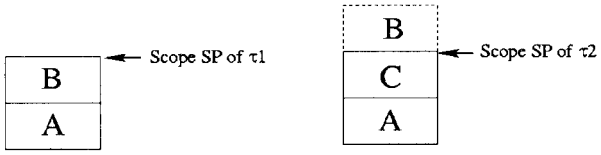
b. τ2 violates the single parent rule..

Figure 1: Violating the single parent rule.

Moreover, if for example, $\tau 1$ enters the region B before $\tau 2$ tries to enter it, $\tau 2$ violates the single parent rule raising the `ScopedCycleException()` exception (see Figure 2.a). But, if $\tau 2$ enters the region C before $\tau 1$ tries to enter it, then it is $\tau 2$ which violates the single parent rule and raises the `ScopedCycleException()` exception (see Figure 2.b). Notice that determinism is an important requirement for real-time applications.



a. τ1 violates the single parent rule.



b. τ2 violates the single parent rule.

Figure 2: Example of non deterministic situation.

3.2 The proposed parentage relation

In order to maintain the single-parent rule of the current RTSJ edition, we consider that the parent of a scoped region is the region within which the region is created [16], and we add the following rules:

1. The parentage relation of regions implies a region tree structure.

2. In the `ScopedMemory` class, the `getOuterScope()` method allows us to know, for the current task the memory region is prior to entering the current region (i.e., its ancestor). This rule was in the former edition of RTSJ.

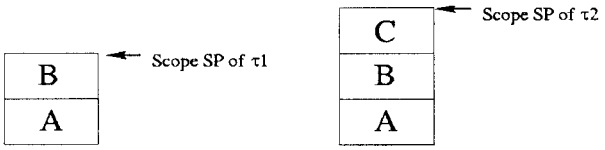
3. Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count of the number of threads having it as current region (*task-counter*), and also a reference count of the number of scoped regions created within the region (*children-counter*).

4. When both task and child reference counters for an instance of the class `ScopedMemory` reach zero, the scoped region is a candidate for reclamation. The *`finalize()`* method of each object allocated within the region must be executed to completion before to collect the region.

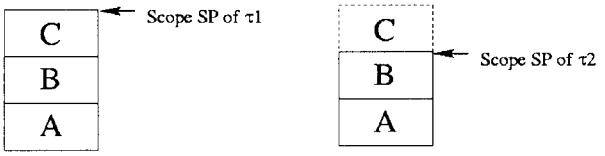
In this way, as in the current RTSJ edition, we obtain a region tree based on a hierarchy relation. But the parent relation is based on the way that scoped regions are created, instead on the order in which scoped regions have been entered by threads.

Consider three scoped regions: A, B, and C, which have been created in the following way: the A region has been created within the heap, the B region has been created within the A region and the C region has been created within the B region. That means that the heap was the current region at the moment of the A object creation, A was the current region at the moment of the B object creation, and B was the current region at the moment of the C object creation. In this way the creation of the A, B, and C scoped regions gives the following parentage relation: the heap is the parent of A, the region A is the parent of B, and B is the parent of C. Then, the child-counter for A and B has been incremented to 1, whereas for C it is 0.

Let us further consider the two tasks $\tau 1$ and $\tau 2$ of our previous example, where we have supposed that task $\tau 1$ has entered areas A and B, which increases by 1 the task-counter for A and B. And task $\tau 2$ has entered areas A and C, which increases by 1 the task-counter for A and C (see Figure 3.*a*). In this situation, the task-counter for A is 2, whereas for B and C is 1. If task $\tau 1$ enters the area C and task $\tau 2$ the area B, at different than those that occur in the suggested implementation of RTSJ [16] [7], the single parent rule is not violated. Then, instead of throwing the `ScopedCycleException()`, we have the situation shown in Figure 3.*b*. At this moment, the task-counter for scoped memory areas A, B, and C are 2.



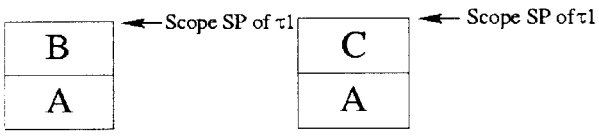
a. $\tau 1$ enters B scoped region and $\tau 2$ enters C.



b. $\tau 1$ enters C scoped region and $\tau 2$ enters B.

Figure 3: The scope stack and the single parent rule.

Note that the scoped stack associated to task $\tau 2$ includes only the A and B scoped regions. Then, even if the task $\tau 2$ has entered the scoped memory C before entering B, pointers from objects allocated in B to objects allocated in C are dangling pointers, as consequence they are not allowed. We consider another situation: task $\tau 1$ enters into scoped area A creates B and C, which increases both the task-counter of A by 1 and its child-counter by 2, whereas both the task-counter and the child-counter of B and C are 0. Then, task $\tau 1$ enters into scoped areas B (Figure 4.*a*) and C (Figure 4.*b*), which increases in 1 the task-counter of both B and C. Only references from objects allocated within B or C to objects within A are allowed. Note that it is not possible for task $\tau 1$ to create a reference from an object within B to an object within C, and vice-versa from an object within B to an object within C, even if task $\tau 1$ must exit the area C before to exit the area B. Then, if a task $\tau 2$ enters into scoped area C and stays there for a while, task $\tau 1$ leaves C and leaves B, the scoped area B can be collected and there are not dangling pointers.



a. $\tau 1$ enters B area. *b. $\tau 1$ enters C area.*

Figure 4: Two state for the coped stack of task $\tau 1$.

Non-scoped areas (i.e. the heap and immortal areas) are not supported in the scoped tree. Moreover, the heap and immortal areas are considered as the *primordial scope*, which is considered to be the root of the area tree [7]. Notice that, for the heap and immortal memory areas, there is no need to maintain the reference-counters because these areas exist outside the scope of the application.

Then, we propose to change the RTSJ specification, so that scoped memory areas are parented at creation time. This new parentage relation introduces great advantages because *i)* simplifies the semantic of scoped memory as the single parent rule becomes trivially true, *ii)* scope cycle exceptions do not occur, *iii)* each thread requires only one scoped stack, and *iv)* the parentage relation does not change during the scoped memory life.

3.3 Scoped region collection

In RTSJ, the method `getReferenceCount()` of the `ScopedMemory` abstract class, allows us to obtain the number of threads that may have access to a scoped region. In this case, when creating a scoped region, the

reference-counter of the region is initialized at zero, and it is increased when entering the region (i.e., through the `enter()` method) or when associating the region or an inner region to a task (i.e., when creating a real-time thread with an initial scoped region through the `RealtimeThread`, `NoHeapRealtimeThread`, or `AsincEventHandler` constructors). And it is decreased when exiting the region (i.e., when returning from the `enter()` method), or when a task leaves the region or an inner region (i.e., when a task using the scoped region exits).

Instead of the method `getReferenceCount()`, we introduce two new methods in the `ScopedMemory` abstract class. That is, `getChildrenCount()` and `getTaskCount()` method, which allows us to know respectively the number of scoped regions created within the considered scoped and the number of tasks for which the region is the current one. Since we suggest maintaining two counters per scoped region, our proposed solution requires taking actions in the following cases:

- When a scoped region becomes the current region for a task (i.e., by entering it or by creating a real-time thread), we must increase the task-reference count of the region. And we must decrease it when the task leaves the region (i.e., when returning from the `enter()` method or when the task exits).

- When creating a scoped region, we must increase the children-reference count of the parent region. And we must decrease it when the created region is collected. Both the children-reference count and the task-reference count of the new region are initialized at zero. And to collect it, we must check that both reference counts reach zero.

Notice that our proposed solution requires to take only one action (i.e., increase/decrease a counter) when a task or a region is created/destroyed, or a region is entered/exited. Whereas in the RTSJ suggested implementation solution, actions are required each time a task is created/destroyed have a *O(n)* complexity, where *n* is the number of nested scoped regions.

3.4 Maintaining the scope stack

In the current RTSJ specification edition, the `enter()` method can throw the `ScopeCycleException()` whenever entering in a scoped region that would violate the single parent rule. The current RTSJ edition also advises to push/pop the entered region on the scope stack belonging to the current task and to increase/decrease the reference counter of the region when the task enters/exits the `enter()` method (see Figure 5). Checking the single parent rule requires an exploration of the scoped stack, in which case we conclude a complexity of *O(n)* for the suggested algorithm of the `enter()` method.

```
enter

if entering ma would violate the single parent rule
        throw ScopedCycleException;
push ma on the scope stack belonging to the current thread;
increase the ma reference count;
execute logic.run method;
decrease the ma reference counter;
pop ma from the scope stack.
```

Figure 5: Current pseudo-code for ma.enter(logic).

The `ScopeCycleException()` method does not appear in the former edition of the RTSJ specification [15], which does not advise using a scope stack associated to each task. Since in our proposed solution there are no cycles in the region tree structure, we do not consider the `ScopeCycleException()`exception, and we avoid the single parent rule checks. In contrast, we consider the region tree to contain all possible scoped stacks associated to all the tasks of an application at a determined instant. Where each scope stack is composed of the current region and all regions following in the path to reach the root of the region tree. The scope stack pointer is part of the task execution context and points to the current region. Figure 6 shows the rewritten pseudo-code for the enter operation, which has constant execution time.

```
enter

make the scope stack pointer points ma in the region tree;
increase the task reference count of ma;
execute logic.run method;
decrease the task reference count of ma;
restore the previous scope stack pointer for the current task.
```

Figure 6: Suggested pseudo-code for ma.enter(logic).

Figure 7 shows the pseudo-code of another operation affecting the scope stack; that is the construction of a new task. In order to maintain the reference counter collector of scoped regions, we must increase/decrease the reference counter of all regions on the scope stack

when creating/destroying the task. Examination of this algorithm reveals a complexity of $O(2n)$.

```
Construct a RealtimeThread

cma = curent memory region;
ima = initial memory region;
if cma is heap or immortal
    create a new scope stack containing cma
else
    start a new scope stack containing the entire current scope stack;
for every scoped memory area in the new scope stack
        increase the reference count;
if ima != current allocation context
        push ima on new scope stack;
run the new thread with the new scope stack;
when the thread terminates
    every memory area pushed by the thread will have been
    popped;
    for every scoped memory area in the scope stack
        decrease the reference count;
free the scope stack.
```

Figure 7: RTSJ pseudo-code to construct a task.

Since we propose a more complex collector for scoped regions based on two reference counters instead of only one, we guarantee the life of scoped regions having children without exploring the stack. Note that an exploration of the scope stack requires a complexity of $O(n)$. Figure 8 describes the suggested behavior for the construction of a real-time thread, which have constant execution time.

```
Construct a RealtimeThread

ima = initial memory region;
make that the new scope stack pointer points ima in the region tree;
increase the task reference count of ima;
run the new thread with the new scope stack;
when the thread terminates decrease the task reference count of ima.
```

Figure 8: Suggested pseudo-code to construct a task.

4 Using a display-based technique

Since assignment rules cannot be fully enforced by the compiler, some dangling pointers must be detected at runtime [7]. Moreover, the RTSJ specification does not explicitly provide an algorithm to enforce the assignment rules. The more basic approach is to take the advice given in the current edition of the RTSJ specification [16], to scan the scoped region stack associated to the current task, verifying that the scoped region from which the reference is created was pushed in the stack before than the region to which the referenced object belongs. This approach requires the introduction of *write barriers* [11]; that is, to introduce a code exploring the scoped region stack when creating an assignment. Note that the complexity of an algorithm which explores a stack is $O(n)$, where n is the depth of the stack.

Since real-time applications require putting boundaries on the execution time of some piece of code, and the depth of the scoped region stack associated with the task of an application are only known at runtime; the overhead introduced by write barriers is unpredictable. In order to fix a maximum boundary or to estimate the average write barrier overhead, we must limit the number of nested scoped levels that an application can hold [10]. We next show how to extend scope tree data structures to perform all required checks in constant time. Our approach is inspired in the suggested parentage relation of scoped memory regions.

4.1 Checking the assignment rules

As stated the RTSJ imposed assignment rules, references can always be made from objects within a scoped memory to objects within the heap or immortal memory; the opposite is never allowed. Also the ancestor relation among scoped memory regions is defined by the nesting regions themselves, and this parentage is supported by the region tree. Since region tree changes occur only at determined moments (i.e., when creating or collecting a scoped region) we can apply the technique based on displays that has been presented in [6]. In this technique, to facilitate constant-time checking for the assignment rules, each scoped region has associated a display containing the type identification codes of its ancestors and its depth in the region tree (see Figure 9).

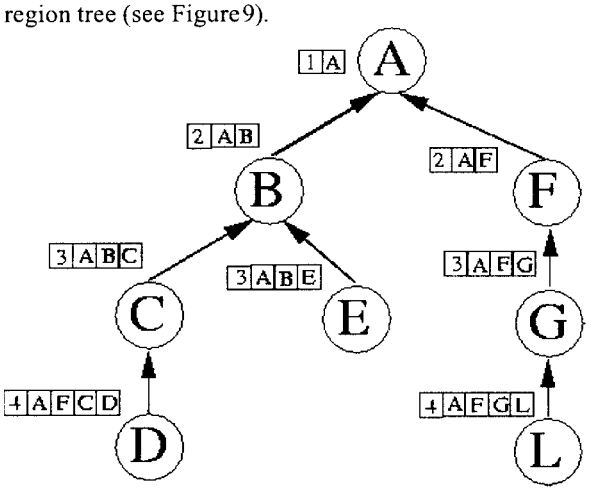


Figure 9: Display-based region tree structure.

In order to use the display-based technique, we suggest including the following rules in RTSJ, instead of the rule which associates a scope stack to each task:

1. The heap and immortal memory regions are always assigned the minimum depth.
2. When a scoped memory region is created, the depth assigned to the new region is the father region depth plus 1.
3. The method `depth()` in the `ScopedMemory` abstract class allows us to know the nested level of the region.

Notice that the depth is the same as the display length, so it is not necessary to store it twice. But to store it allows us to have a more efficient implementation and a simpler exposition of the algorithm[6]. Figure 10 shows the pseudo-code that we must introduce in the execution of each assignment statement (e.g., **`x.a=y`**) to perform the assignment checks in constant-time.

Write barrier

```
X = region to which the x object belongs;
Y = region to which the y object belongs;
if ((Y.depth <> 0) and (X.display[Y.depth]<>Y.display[Y.depth]))
        illegalAssignment();
```

Figure 10: Checking the assignment rules.

4.2 Maintaining the display structure

This parentage relation is less dynamic than in the current RTSJ edition, where the parent-child relations changes as scoped memory regions are entered and exited. Thus, the associated type hierarchy is not fixed, but only changes at the point that the children reference count increases or decreases. Then, the management of displays only requires to copy the parent display including the new created region identifier at the end of it when creating a scoped region, and to invalidate it when the region is collected. Then, the structure of the display tree does not affected, when entering/exiting a region or creating/destroying a thread.

The current RTSJ edition [16] also presents another method allowing a task to change the allocation context; the `executeInArea()` method, which checks the current scope stack in order to find the region to which the message associated with the method is sent. If it is found, a new scope stack containing the found region and all scopes below this region on the scope stack is started. If it is not found, the `InaccesibleAreaException()` exception is raised. In this method, the `newArray()` and `newInstance()` methods allow a task to allocate objects outside the current region. Since these methods require an exploration of the stack, they have an *O(n)* complexity (see Figure 11).

executeInArea, newArray or newInstance

```
if ma is an instance of heap or immortal
        start a new scope stack containing only ma
else ma is scoped
        if ma is in the scope stack for the current task
                start a new scope stack containing ma and all
                scopes below ma on the scope stack
        else throw InaccessibleAreaException;
make the new scope stack the scope stack for the current thread;
execute logic.run or construct the object;
restore the previous scope stack for the current thread;
discard the new scope stack.
```

Figure 11: Current memory context switch.

Notice that in our proposed solution entering a region older than the current one that is in the same branch of the region tree (i.e., in the same scope stack), has the same consequences as the `executeInArea()` method. Therefore, this method is not strictly necessary, and actually it does not appear in the former edition of the RTSJ specification [15]. Figure 12 shows the rewritten pseudo-code using displays, which is constant-time executable.

executeInArea, newArray or newInstance

```
cma = current region
if ((ma.depth <> 0) and  (cma.dispaly[ma.depth] <>
ma.dispaly[ma.depth]))
        throw InaccessibleAreaException;
make  ma the current region;
execute logic.run  or construct the object in ma;
restore the previous current region.
```

Figure 12: Memory context switch with displays.

4.3 Estimating the write barrier overhead

We have modified the KVM [17] to implement three types of memory regions: *(i)* the heap that is collected by the KVM GC, *(ii)* immortal that is 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 `ImmortalPhysicalMemory` class has not been implemented.

To obtain the overhead that write barrier introduces , two measures are combined: the number of events, and the cost of the event. We use an artificial collector benchmark which is an adaptation made by Hans Boehm from the John Ellis and Kodak benchmark[4]. This benchmark executes $262*10^6$ bytecodes and allocates 408 Mbytes. The number of executed bytecodes performing the 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 `putstatic_fast`: 0). This means that 5% of executed bytecodes perform a write barrier test, as already obtained with SPECjvm98 in [18].

The write barrier cost is proportional to the number of executed evaluations. With our proposed solution, the overhead introduced to evaluate a condition of the write barrier test in the KVM is about 16% in each assignment. Because of this, the average write barrier cost introduced in an application is only 1.6%. But the most important consequence of this approach is that the time taken to detect an allowed or dangling reference is the same, and it does not depend on the nested level of the region to which the two objects of the memory reference belong.

5 Related works

The main contribution of our approach is to avoid the single parent checks by changing the parentage relation of scoped region within the region tree, which ensures all algorithms managing scoped regions to be executed in constant-time. A study of the behaviour of the RTSJ simple parent rule and a first approach in order to avoid checking it when entering a scoped area has been presented in [12]. Our proposed solution also simplifies the maintaining of the display structure used to check illegal assignments in [6]. The display-based technique has been introduced in [5] to evaluate expressions in constant time. And it was firstly used to support RTSJ scoped region in [6]. The main difference between both techniques is that the encoding of the type hierarchy in [5] is known at compile time, whereas in [6] the region tree changes at runtime. Our proposed solution makes the scoped region tree structure more static, because the region tree structure changes only when a scoped region is created/destroyed instead every time a thread enters/exits a scoped region. An alternative technique to subtype test in Java have been presented in [14]. This technique has been extended to perform memory access checks in constant-time.

The idea of using both write barrier and a stack of scoped regions ordered by life-times to detect illegal inter-region assignments was first introduced in [9]. A similar approach using also a stack-based memory management that operates dynamically is given in [3].

The most common approach to implement read/write barriers is by inline code, consisting in generating the instructions executing barrier events for every load/store operation. Beebe and Rinard use this approach [1], and their implementation uses five runtime heap checks to ensure that a critical task does not manipulate heap references. Alternatively, our solution instruments the bytecode interpreter, avoiding space problems, but this still requires a complementary solution to handle native code [11]. The use of the hardware support for write barriers has been the subject of [8] and [9], where we propose to improve the performance of checking illegal references by two different ways: using existing hardware support and modifying existing hardware. The performance improvement introduced by these solutions has been compared in [10].

In [4], we found a region-based approach to memory management in Java based on static analysis. But the dynamic issues that Java presents, requires for some cases to check the assignment rules at run-time. However, static and dynamic techniques can be combined to provide more robustness and predictability of RTSJ applications.

6 Conclusions

To enforce the RTSJ imposed rules, a compliant JVM must check both the single parent rule on every attempt to enter a scoped memory region, and the assignment rules on every attempt to create a reference between objects belonging to different memory regions. Since objects references occur frequently, it is important to implement checks for assignment rules efficiently and predictably.

To support scoped memory regions, we propose a mechanism based on a reference-counter collector and a scoped region tree based on the parentage relation of scoped regions, which contains all scope stacks allowed in the system in a given instant. Note that by collecting regions, problems associated with reference-counting collectors are solved: the space and time to maintain two reference-counts per scoped region is minimal, and there are no cyclic scoped region references. The parentage relation is based on the way they are created/collected, instead of the way they are entered/exited by tasks.

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

Every scoped region has two reference counters associated to it, which allows us a more efficient management of regions, making it time predictable. Every region has also a scope stack supporting all scoped regions containing objects allowed to be referenced from it. When a task enters a region, the region stack associated to the region becomes the scoped region associated to the task. The scope stack can be coded as a display, which allows us to use subtype test based techniques making the enforcement of memory references time-predictable. It is interesting to compare the proposed solution and the hardware-based solutions.

References

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

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

[3] D.J. Cannarozzi, M.P. Plezbert, and R.K. Cytron. "Contamined Garbage Collection". In Proc. of the Conference of programming Languages Design and Implementation (PLDI). ACM SIGPLAN, May 2000.

[4] M. Christiansen, P. Velschow. "Region-Based Memory Management in Java". Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

[5] N.H Cohen. "Type-Extension Type Tests Can Be Performed in Constant Time". ACM Transactions on Programming Languages and Systems (TOPLAS), 1991.

[6] A. Corsaro and R.K. Cytron. "Efficient Reference Checks for Real-time Java".ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems", LCTES 2003.

[7] P.C. Dibble. "Real-Time Java Platform Programming". Prentice Hall 2002.

[8] 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, January 2004.

[9] 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.

[10] M.T. Higuera and, V. Issarny "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.

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

[12] M.T. Higuera-Toledano. "Studying the Behaviour of the Single Parent Rule in Real-Time Java". In Proc of 2on Workshop on Real-time Java (JTRES), 2004.

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

[14] K. Palacz and J. Vitek. "Java Subtype Tests in Real Time" In Proc of 17th European Conference for Object-Oriented Programming, (ECOOP) 2003.

[15] The Real-Time for Java Expert Group. "Real-Time Specification for Java". Addison-Wesley, 2000.

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

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

[18] Standard Performance Evaluation Corporation: SPEC Java Virtual Machine Benchmark Suite. http://www.spec.org/osg/jvm98, 1998.