Improving the Scoped Memory Region Garbage Collector of Real-Time Java[*]

M. Teresa Higuera-Toledano

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

Email: mthiguer@dacya.ucm.es

Abstract. This paper addresses the issue of improving the performance of memory management for real-time Java applications, building upon the Real-Time Specification for Java (RTSJ). In order to avoid cycles among scoped memory regions, RTSJ introduces the single parent rule, which requires to be checked at run-time. The problem here is that the parentage relation as it is formulated by RTSJ introduces an unfamiliar programming model, and a high time overhead on the program execution that it is no possible to determine and must be bounded. From our point of view, the single parent rule adversely affects both the performance and predictability of the RTSJ application. This paper presents an efficient algorithm for managing the scoped regions garbage collector, which requires modification of the single parent rule definition.

1 Introduction

From a real-time perspective, the Garbage Collector (GC) introduces unpredictable pauses that are no tolerated by real-time tasks. Real-time collectors eliminate this problem but introduce a high overhead. The memory model used in the *Real-Time Specification for Java* (RTSJ) [6] offers real-time guarantees to critical tasks by tuning of the GC. RTSJ provides two alternatives to using the garbage collected heap; the *immortal* memory and *scoped* memory. In order to prevent the creation of dangling pointers, RTSJ imposes strict assignment rules to or from memory regions. Given that one of the requirements for RTSJ is that there should be no changes in the Java language and that Java base line compilers can be used to compile RTSJ programs, these rules must be enforced at run-time. This solution adversely affects both the performance and predictability of the RTSJ application. The success of the RTSJ may well hinge on the possibility to offer an efficient and time-predictable implementation of scoped memory regions.

[*] 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.

Since scoped memory regions can be nested, the full implication of the memory assignment rules must be considered and studied in detail. On the current RTSJ memory model, the memory assignment rules by themselves do not prevent the dangling reference problem. To avoid this problem, RTSJ introduces the single parent rule, which requires that each scoped memory region has a single parent. The problem here is that the parentage relation as it is formulated by RTSJ introduces an unfamiliar programming model, and a high time overhead on the program execution that it is no possible to determine and must be bounded. More over, the program behavior of RTSJ programs can present race carrier conditions.

From our point of view, RTSJ requires reevaluation and possible redesign the memory management rules and suggested algorithms to implement a real-time virtual machine. In this paper we present an efficient algorithm for managing scoped memory which requires some modifications in the current RTSJ specification. Particularly, we propose to reformulate the single parent rule. Then, we propose more natural parentage relation for scoped memory regions, which gives a more natural programming model, avoids race carrier conditions, reduces highly the introduced overhead, and makes time-predictable the execution program.

This paper focuses on data structure and algorithms supporting the RTSJ single parent rule. First, we present an in depth description of the RTSJ memory model and the problems associated to the single parent rule (Section 2). We suggest a solution to improve the RTSJ memory model, particularly the scoped region collection, which can be implemented efficiently by using simple data structures and algorithms (Section 3). Finally a summary of our contribution conclude this paper (Section 5).

2 The RTSJ memory model and the non-deterministic behavior

There is only one object instance of the heap and the immortal region in the system, which are resources shared among all threads in the system. In contrast, for scoped regions several instances can be created by the application. Several related real-time threads, can share a memory region, and the region must be active until at least the last thread has exited. The default memory region is either the heap or the immortal memory region. Also, the initial default memory allocation region of a real-time thread can be specified when the thread is constructed. The active region associated with the real-time thread change when executing the `enter()` method, which is the mechanism to activate a region. Since 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. These rules avoid references from an object to another one with a potentially shorter lifetime. Then, we must ensure that the following conditions are checked before executing an assignment:

- Objects within the heap or an immortal region cannot reference objects within a scoped region.
- Objects within a scoped region cannot reference objects within a scoped region that is not the same or an outer one.

Illegal assignments must be checked when executing instructions that store references within objects or arrays. The `IllegalAssignment()` exception throws when detecting an attempt to make an illegal pointer. Since assignment rules cannot be fully enforced by the compiler, some dangling pointers must be detected at runtime, which requires the introduction of write barriers [3]. That is, to introduce a code checking for dangling pointers when creating an assignment.

Also, a safe region implementation requires that a region gets deleted only if there is no external reference to it. This problem has been solved by using a reference-counter for each region that keeps track of the use of the region by threads (i.e., a depth counter), and a simple reference-counting GC collects scoped memory regions when their counter reaches 0.

2.1. The nested region stack and the single parent rule

In order to keep track of the currently active memory regions of each schedulable object, RTSJ uses a stack associated which each real-time thread. Every time a real-time thread enters a memory region, the identifier of the region is pushed onto the stack. When the real-time thread leaves the region, its identifier is popped of the stack. The stack can be used to check for illegal assignments among scoped memory region (we consider that the stack grows up):

- A reference from an object X within a scoped region A to another object Y within a scoped region B is allowed whether the region B is below the region A on the stack.
- All other assignment cases among scoped regions (i.e., the region B is above the region A or it is not on the stack) are forbidden.

Note that it can appear cycles among scoped regions on the region-stack. For example, if both scoped regions A and B appears on the following order: A, B, A, then are allowed both reference types: from A to B, and from B to A. That means that the A scoped region is inner to the A scoped region, and vice-versa. Since the assignment rules and the stack-based algorithm by themselves does not enforce safety pointers, the RTSJ defines the single parent rule, which goal is to avoid scoped region cycles on the stack [2]:

"If a scoped region is not in use, it has no parent. For all other scoped objects, the parent is the nearest scope on the current entered scoped region stack. Except for the primordial scope, which represents heap, immortal and immortal physical memory, only scoped memory regions are visible to the single parent rule. A scoped region has exactly zero or one parent."

This parentage relation guarantees that once real-time thread has entered a set of scoped regions in a given order, any other real-time thread will have to enter them in the same order. At this time, if the scope region has no parent, then the entry is allowed. Otherwise, the real-time thread entering the scoped region must have entered every proper ancestor of it in the scope stack.

2.3. The overhead and non-deterministic behaviour problems

There are four operations affecting the scope stack and the reference-counter collector. Particularly the reference-counter of a scoped memory region is increased when entering a new inner scoped through the `enter()` method (i.e., the reference count of all scoped regions on the current stack must be increased). And it is decreased when returning from the `enter()` method, when the real-time thread using the scoped region exits, or when an inner scope returns from it's `enter()` method (i.e., the reference count of all scoped regions on the current stack must be decreased). When the reference-counter of a scope region is zero, a new nesting (parent) for the region will be possible.

Since the suggested algorithms implementing the methods affecting the scoped region collector require an exploration of the stack, they have a complexity of O(n), where n is the depth of the stack. Note that it is possible for a scoped region to have several parents throughout its life, which results in an unfamiliar programming model. Also, the single parent rule can result in race carrier conditions, which gives a non deterministic behaviour to RTSJ programs [4], [5]. As an example, consider two real-time threads T1 and T2. Where the real-time thread T1 enters regions in the following order: A and B, whereas T2 enters regions as follows: B and A. Let us suppose that T1 and T2 have entered respectively the A and B regions and both stay there for a while. In this situation, the application has two different behaviours:

- When T1 tries to enter the B scoped region, it violates the single parent rule.
- When T2 tries to enter the A scoped region, it violates the single parent rule.

Then the `ScopedCycleException()` exception throws by four different conditions. More over, the single parent rule is not violated and the application gives the correct result in other possible cases (e.g., T1 enters A and B, and exits both regions before T2 enters B and A).

3. Studying an alternative approach

The RTSJ parentage relation is not trivial: there are *orphan* and *adopted* regions. When a region is not in use it has no parent, and the parent of a region can change along its life, which results in an unfamiliar programming model. More over, the same program can have several results depending on race carrier conditions. Another source of indeterminism is the stack: the introduced overhead by the algorithms exploring the stack is high and unpredictable[1]. Real-time applications require putting boundaries on the execution time of some piece of code. Since the depth of the stack associated with the real-time threads of an application are only known at runtime, to estimate the average write barrier overhead, we must limit the number of nested scoped levels that an application can hold. This unpredictability can make it impossible to establish bounds for the time taken by service requests in distributed real-time Java solutions [1].

[1] Each time a real-time thread is created/destroyed, enters/exits a region, or executes the executeInArea() or newInstance() method, requires the execution of a stack-based algorithm.

3.1. Redefining the parentage relation

In order to avoid race carrier conditions, we propose to change the parentage relation among memory regions as follows:

“The parent of a scoped memory region is the memory region in which the object representing the scoped memory region is allocated”

Note that the parent of a scoped region is assigned when creating the region and does not change along the life of the region. As consequence there are no orphan regions, or adopted regions by several times. Consider two scoped regions A and B created within the heap. That means that the heap is the parent of both scoped regions A and B. As different that occurs in RTSJ, pointers from objects within A to objects within B and vice-versa are disallowed:

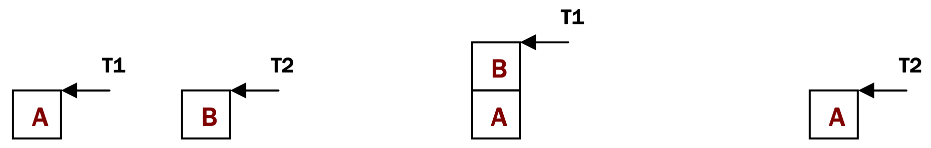
- If T1 enters both regions A and B, and then T2 enters B, the single parent rule is not violated. Instead of throwing the `ScopedCycleException()`, both scope stacks, thus associated to T1 and thus associated to T2 include only the B region(see Figure 1.*a*).
- If T2 enters both regions B and A, and then T1 enters A, the single parent rule is not violated. Instead of throwing the `ScopedCycleException()`, both scope stacks, thus associated to T1 and thus associated to T2 include only the A region (see Figure 1.*b*).

T1
T2
B
B
T1
T2
A
A
a. T1 enters A and B, then T2 enters B
b. T2 enters B and A, then T1 enters A

Fig 1. While references from B to A are allowed, from A to B are illegal.

Note that with this parentage relation, illegal references are known before to run the program and there are not race carrier conditions.

As another example, we consider two scoped regions: A and B, which have been created in the following way, the A region has been created within the heap, and the B region has been created within the A region. Then, the creation of the A and B scoped regions gives the following parentage relation: the region A is the parent of B. Let us further consider two real-time threads T1 and T2, where T1 and T2 have entered respectively regions A and B (see Figure 2.*a*), if T1 enters B (see Figure 2.*b*) or T2 enters A (see Figure 2.*c*) the single parent rule is not violated. Even if T2 has entered B before entering A, references from objects allocated within A to objects allocated within B are dangling pointers, as consequence are illegal. Regarding assignment rules, we found no problem for pointers from B to A created as consequence of the execution of T1, T2, or any other real-time thread on the system. This situation is stable independently of the real-time thread that makes the reference.



a. T1 enters A and T2 enters B *b. T1 enter sB* *c. T2 enters A*

Fig 2. While references from B to A are allowed, from A to B are illegal.

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

- When a scoped region becomes the current region for a real-time thread (i.e., by entering it or by creating a real-time thread), we must to increase the task-count of the region. And we must decrease it when the thread leaves the region (i.e., when returning from the `enter()` method or when the thread exits).
- When creating a scoped region, we must increase the children-count of the parent region. And we must decrease it when the created region is collected. Both the children-count and the task-count of the new region are initialized at zero.
- To collect a scoped region, we must check that both reference counts: the children-count and the task-count reach zero.

Whereas in RTSJ, actions are required each time a real-time thread is created/destroyed or a region is entered/exited; having a *O(n)* complexity, where *n* is the number of nested scoped regions. In our proposed solution requires only one action (i.e., increase/decrease a counter) when a real-time thread or a region is created/destroyed, or a region is entered/exited.

As an example, we consider two scoped regions: A, and B, which have been created in the following way, the A region has been created within the heap, and the B region has been created within the A region. Then, the child-count for A has been incremented to 1, whereas for B it is 0. Let us further consider the two real-time threads T1 and T2, where T1 enters regions A and B, which increases by 1 the task-count of both A and B (see Figure 3.*a*). T2 enters regions B and A, which increases by 1 the task-count of both regions B and A (see Figure 3.*b*). Then T1 exits B and A (see Figure 3.c), and T2 exits A. In this situation, the task-count for A is 0 (see Figure 3.*d*). Since references from objects within B to objects within A are allowed, the region A must not be collected. When T2 exits B, the task-count of B reach zero (see Figure 3.*e*), then B can be collected, which makes A collectable (see Figure 3.f).

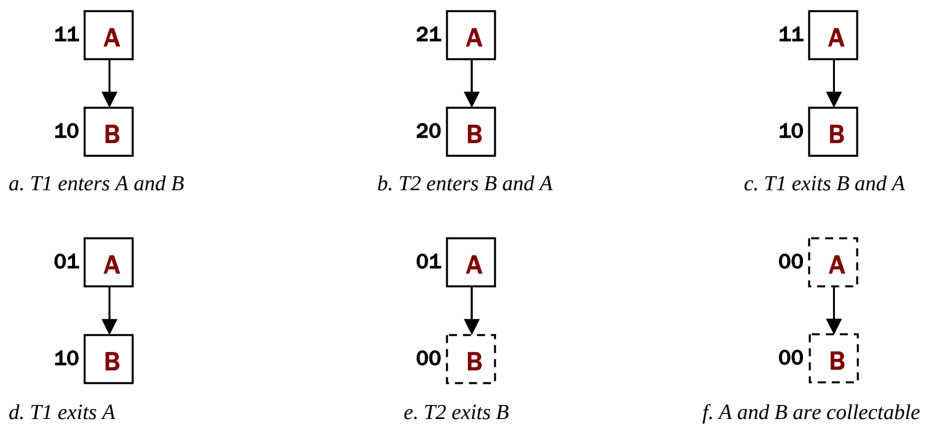


Fig 3. The task-counter and children-counter of scoped regions A and B.

We consider another situation: the real-time thread T1 enters into the scoped region A and creates the B and C scoped regions, which increase both the task-count of A by 1 and its child-count by 2, whereas both the task-count and the child-count of B and C are 0 (see Figure 4.*a*). Then, T1 enters into scoped regions B and C, which increases in 1 the task-count of both B and C (see Figure 4.*b*). Then, if another real-time thread T2 enters into scoped region C, and stays there for a while (see Figure 4.*c*), T1 leaves C and leaves B (see Figure 4.*d*), the scoped region B can be collected and there are no dangling pointers (see Figure 4.*e*). At this moment, the C region can not be collected because its task-count is 1, and the A region can not be collected because its children-count is 1. When T2 exits C, the task-count of C reaches 0, which makes C collectable and decreases the children-count of A (see Figure 4.*f*), which reaches zero making A collectable.

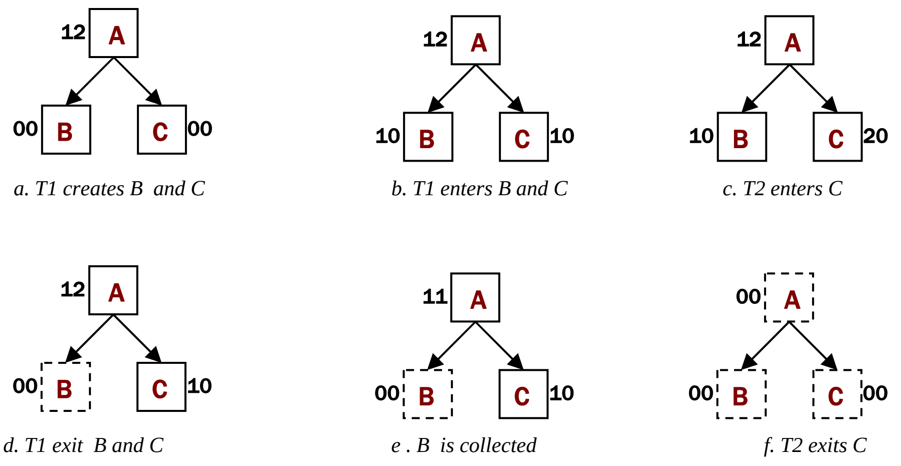


Fig 4. The task-count and children-count of scoped regions A, B, and C.

4 Conclusions

To enforce the RTSJ imposed rules, a compliant JVM must check the single parent rule on every attempt to enter a scoped memory region, it is important to implement checks for assignment rules efficiently and predictably. In our proposed solution, the parentage relation of regions is based on the way they are created/collected, instead of the way they are entered/exited by tasks such as the RTSJ suggests. Then, we avoid increasing/decreasing the reference counter of all memory regions on the stack every time that a real-time thread enters/exits a scoped memory region.

In order to do the implementation of required algorithms more efficient, every scoped region has two reference counters, which allows us a more efficient management of regions, making it time predictable. Note that by collecting memory 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. Note that the introduction of this change in the parentage relation simplifies the complex semantics for scoped memory regions adopted by RTSJ.

As an inconvenience of this approach, is that it is less expressive than RTSJ. Consider a real-time thread that enters into scoped region A and creates both B and C. Then, T1 enters into B and next into C. But, at different that occurs in RTSJ, it is not possible for T1 to create a reference from an object within B to an object within C; even if T1 must exit the region C before to exit the region B.

References

1. 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.
2. P.C. Dibble. "Real-Time Java Platform Programming". Prentice Hall 2002.
3. M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Area-based Memory Management for Real-time Java". In Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2001.
4. M.T. Higuera-Toledano. Towards an Understanding of the Behavior of the Single Parent Rule in the RTSJ Scoped Memory Model. In Proc of 11th Real-time and Embedded Technology Application Symposium (RTAS). IEEE 2004.
5. 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.
6. The Real-Time for Java Expert Group. "Real-Time Specification for Java". RTJEG 2005. http://www.rtsj.org