The Indeterministic Behavior of Scoped Memory in Real-Time Java*

M. Teresa Higuera-Toledano

Facultad Informática, Universidad Complutense 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) includes both a heap within a traditional garbage collector, which collects the not used objects, and a new memory management feature based on scoped memory areas. The scoped memory areas model allows programmers to ensure constant-time reclamation thus to have predictable performance. In order to maintain the pointer safety of Java, RTSJ imposes strict assignment rules to or from memory areas preventing the creation of dangling pointers, at the cost of an unfamiliar programming model. The guidelines given by RTSJ to implement the assignment rules also increase the program complexity; more over, makes indeterminist the program behavior. In order to solve these problems, we propose to redefine some RTSJ rules.

Keywords: Real-time Java, Scoped-regions, Single parent rule, Illegal assignments, Garbage collection, Write-barriers.

1. Introduction

The *Real-Time Specification for Java* (RTSJ) [11] allows us to use the Java environment in the construction of real-time systems. The interdependence between functional and real-time semantics of real-time software makes its maintenance especially difficult. In addition, embedded software systems are not portable as they depend on the particular underlying operating system and hardware architecture. The Java environment provides attributes that make it a powerful platform to develop embedded real-time applications. Since embedded systems normally have limited memory, an advantage that Java presents is the small size of both the Java runtime environment and the Java application programs. Dynamic loading of classes facilitates the dynamic evolution of the applications embedded in the system. Additionally, the Java platform provides classes for building multithreaded applications and automatic garbage collection. However, it does not guarantee determinism nor bounded resource usage, which this type of systems needs.

In order to solve the shortcomings of Java regarding its use for embedded real-time programming some solutions are been proposed. In [5], we review the proposed real-time Java solutions, considering and analyzing the following embedded real-time issues: *i)* Solutions to access the underlying hardware. *ii)* What is the most adequate model for real-time thread scheduling, *iii)* How to solve the priority inversion problem, *iv)* How to generate and handle asynchronous events, *v)* How to manage resources. And *vi)* how to modify the garbage collection in order to make it compatible with real-time tasks execution. From our point of view, RTSJ [11] constitutes the most adequate solution for real-time systems, which use in mission critical systems are currently been evaluated in a number of projects such as [3].

RTSJ covers well all aforementioned features, in particular, the memory model includes both a heap with a traditional garbage collector, and a new memory management feature based on scoped memory areas. From a real-time perspective, the garbage collector introduces unpredictable pauses that are not tolerated by real-time tasks. Real-time collectors eliminate this problem but introduce a high overhead. The scoped memory areas model allows programmers to ensure constant-time reclamation thus to have predictable, at the cost of its explicit management which affects particularly how programs are written.

Because scoped areas can be reclaimed at any time, objects within a memory area with a longer lifetime are not allowed to create a reference to an object within another area with a potentially shorter lifetime. An RTSJ

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

implementation must enforce these scope checks before executing an assignment. In order to do that, RTSJ establishes a parentage relationship between the scoped memory areas; which is called the *single-parent* rule. To enforce the RTSJ imposed rules, a compliant JVM must assure both the single parent rule and the assignment rules. The suggested RTSJ implementation requires checking the single parent rule on every attempt to enter a scoped memory area, and to explore the scope stack on every attempt to create a reference. Since objects references occur frequently, it is important to implement checks for assignment rules efficiently and predictably.

In this paper, we avoid the exploration on the scope stack, by replacing it by a name-based technique making the enforcement of memory references time-predictable, because it does not depend of the nested level of the area to which the two objects of the memory reference belong. In order to do that, we propose to base the parentage relation of memory areas on the way they are created/collected, instead on the way they are entered/exited by tasks such as the RTSJ suggests. More over, we avoid checks on every attempt to enter a scoped memory area.

1.1 Related work

The main contribution of our approach is to introduce a name-based solution for illegal assignments, which avoid the exploration of the scope stack as suggest the current RTSJ. This solution is a direct consequent of the work presented in [7], which shows how all the necessary run time checks can be performed in constant time by simplifying the scoped memory hierarchy. This allows us to use a name-based encoding to implement dynamic scope checks, and also to avoid the single-parent rule checks. In [12] we present a study of the behaviour of the RTSJ simple parent rule and a first approach in order to change the parentage relation of scoped memory areas, avoiding its checks when entering a scoped area.

The work presented in [1] introduces a display-based technique to support RTSJ scoped area and to check illegal assignments. An alternative technique to subtype test in Java have been presented in [10], which has been extended to perform memory access checks in RTSJ. Another solutions tray to maintain statically the RTSJ invariants such as [9] and [13]. However, the dynamic issues that Java presents, requires 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.

1.2 Paper Organization

The paper is organized as follows. Section 2 presents an in depth description of the semantics of the RTSJ memory model, being centred in the scoped memory areas and its relations, giving sufficient details about the subtleties of this model. Section 3 introduces an alternative solution to improve the RTSJ suggested memory model implementation, which is based on the identifier of memory areas and includes a new point of view in the way to understand the RTSJ memory area relationships, particularly the single-parent rule. Section 5, finally, concludes this paper.

2 The RTSJ memory model

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. This must not lead to undertake the unsafe primitive solution that consists in letting the application programmer to explicit deal with memory reclamation. An alternative approach is to use memory regions within which both allocation and de-allocation are customized, also space locality is improved. Such a facility is supported by RTSJ through the introduction of three kinds of regions, called memory areas.

2.1 The memory areas model

The RTSJ introduces memory regions and allows the implementation of real-time compliant collectors to be run within regions except within those associated with hard timing constraints. The `MemoryArea` abstract class provides three kinds of regions having different properties in term of both the object lifetimes and the object allocation/de-allocation timing guarantees:

i) Immortal memory areas contain objects whose life ends only when the JVM terminates and are never garbage collected.

ii) Scoped memory areas, supported by the `ScopedMemory` abstract class, enables grouping objects having well-defined lifetimes. Scoped areas are collected when there is not a thread using the area, and may either

offer temporal guarantees or not on the time taken to create objects.

iii) The garbage collector within the *heap* must scan all objects allocated within immortal or scoped memory areas for references to any object within the heap in order to preserve the integrity of the heap. RTSJ further defines the `GarbageCollector` abstract class, which can be customized through an incremental collector allowing the application to execute while the collector has been launched.

Each memory area is then managed to embed objects that are related regarding associated lifetime and real-time requirements. Particularly, objects allocated within immortal memory areas live until the end of the application and are never subject to garbage collection. Objects with limited lifetime can be allocated into a scoped area or the heap. Garbage collection within the application heap relies on the (real-time) collector of the JVM.

2.2 The task model

RTSJ makes distinction between three main kinds of tasks: *i) low-priority* that are tolerant with the garbage collector, *ii) high-priority* that cannot tolerate unbounded preemption latencies, and *iii) critical* that cannot tolerate preemption latencies. Then, RTSJ introduces two new kind of thread; both are real-time threads. Nevertheless, the latter is protected from the collector delays, which can come about for two reasons:

- The collector is invoked during the execution of a real-time thread as consequence of memory allocation.
- The collector is running and must arrive to a point where all data structures are in a consistent state to be preempted by a high-priority task (*preemption latency*), which can be also cause the *inversion priority problem.*

In RTSJ critical tasks avoids both problems by running at a higher priority than the collector does, also they are not allowed to access heap allocated objects (i.e., critical tasks do not cause heap allocation and do not require heap allocated objects). An application can allocate memory into memory areas, as follows:

i) Low-priority tasks or traditional threads can allocate memory only within the traditional heap.

ii) High-priority tasks or real-time threads may allocate memory within the heap or within a memory area other than the heap by making that area the current allocation context (e.g., by entering the area).

iii) Critical tasks or non-heap real-time threads must allocate memory from a memory area other than the heap by making that area the current allocation context.

A new allocation context is entered by calling the `MemoryArea.enter()` method or by starting a real-time thread whose constructor was given a reference to a memory area. As an example, Figure 1 shows a real-time thread called myTask (line 13), which allocates an array of 10 integers within the heap (line 5), and another of 20 integers in the scoped memory area called myRegion. Once a memory area is entered (line 15), subsequent uses of the `new` keyword, within the program logic, will allocate objects from the memory context associated to the entered area (line 6). When the area is exited, subsequent uses of the `new` operation will allocate memory from the area associated with the enclosing scope.

```
1:  import javax.realtime;
2:
3:  class Allocator implements Runnable {
4:  public void run() {
5:          HeapMemory.instance().newArray(Integer, 10);
6:          int[] x = new int[20];
7:      }
8:  }
9:
10: class RegionUseExample {
11:     public static void main (String[] args) {
12:         ScopedMemory myRegion = new VTMemory(1024, 2*1024);
13:         RealtimeThread myTask = new RealtimeThread(null, null,
14:                                     new MemoryParameters(1024, 0),
15:                                     myRegion, null,
16:                                     new Allocator());
17:          myTask.start();
18:     }
19: }
```

Figure 1: Using RTSJ memory regions.

2.3 The scoped memory areas

Scoped regions may or may not be subject to internal real-time garbage collection depending on their temporal properties. However, since RTSJ does not impose the collection of objects within scoped regions, we consider in this paper that scoped regions are never garbage

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

collected. Since objects within immortal and scoped areas are not garbage collected, they may be exploited by critical tasks. A scoped region gets collected as a whole once it is no longer used. The lifetime of objects allocated in scoped areas is governed by the control flow. Strict assignment rules placed on assignments to or from memory areas 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.

An implementation solution to ensure the checking of these rules before each assignment statement consists of performing it dynamically, each time a reference is stored in the memory (i.e., by using write barriers). This solution adversely affects both the performance and predictability of the RTSJ application.

2.4 The single parent rule

Scoped areas can be nested and each scope can have multiple sub-scopes. Several related threads, possibly real-time, can share a memory area, and the area must be active until at least the last thread has exited. When no active threads within the scoped area, the entire memory assigned to the area can be reclaimed along with all objects allocated within it. The RTSJ suggested implementation associates to each real-time thread a *scope stack* containing all the areas that the thread has entered but not exited. The structure of enclosing scopes (i.e., the scope stack) is accessible through a set of methods on the `RealtimeThread` class, which allows outer scopes to be accessed like an array. In order to maintain the scope stack contain all nested scoped areas that a thread can hold, RTSJ establishes the *single parent rule*:

"If a scoped region is not in use, it has no parent. For all other scoped objects, the parent is the nearest scope outside it on the current scoped region stack. A scoped region has exactly zero or one parent."

The parentage relationship requires that a scoped memory area has exactly zero or one parent. Scoped areas 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. 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 area to be almost once in the scope stack associated with the task.

The single-parent rule also enforces every task that uses a memory area to have exactly the same scoped area parentage. Consider two scoped memory areas, A and B, where the A scoped area is parent of the B area. In such a case, a reference to the A scoped area 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 within B raises the `IllegalAssignment()` exception.

Since scoped areas are collected when there is not a thread using the area, each scoped memory area object (i.e., each instance of the class `ScopedMemory`) must maintain a reference count of the number of threads in which it is being used. When the reference count for a scoped area is decreased from one to zero, all objects within the area are considered unreachable and are candidates for reclamation.

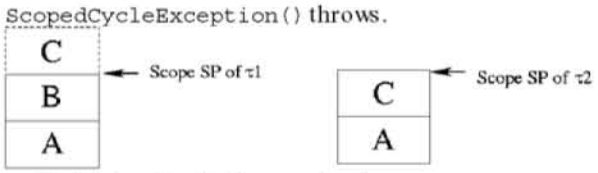
3 The name-based solution

We suppose that the most common RTSJ use of a scope area is repeatedly to perform the same computation in a periodic task. In the current RTSJ, when a task or an event handler tries to enter a scoped area S, we must check if the corresponding thread has entered every ancestor of the area S in the scoped area tree. Then, safety of scoped areas requires checking both the set of rules imposed on their entrance and the aforementioned assignment rules. Both tests require algorithms, the cost of which is linear or polynomial in the number of memory areas that the task can hold. In order to optimize the RTSJ memory model, we suggest simplifying data structures and algorithms, and propose to change the definition of the single parent rule.

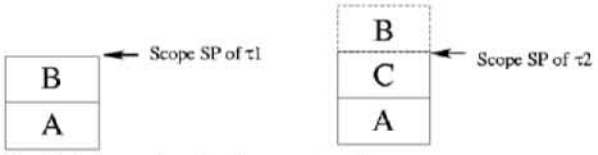
3.1 The indeterminism of single parent rule

The implementation of the single-parent rule as suggests the current RTSJ edition [11] 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 areas in a given order, any other thread is enforced to enter the set of areas in the same order. Consider three scoped areas:

A, B, and C, and two task τ1 and τ2. Where task τ1 tries to enter the areas as follows: A, B, and C, whereas τ2 tries to enter the areas in the following order: A, C and B. Let us suppose that task τ1 has entered areas A and B, and task τ2 has entered areas A and C. If task τ1 tries to enter the area C (see Figure 2.a) or task τ2 tries to enter the area B (see Figure 2.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 2: Violating the single parent rule.

Moreover, if for example, τ2 enters the area C before τ1 tries to enter it, then it is τ2 which violates the single parent rule and raises the `ScopedCycleException()` exception (see Figure 3.a). However, if τ1 enters the area B before τ2 tries to enter it, τ2 violates the single parent rule raising the `ScopedCycleException()` exception (see Figure 3.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 3: Example of non-deterministic situation.

3.2 The proposed parentage relation

In order to solve the indeterminism problem introduced by scoped memory in RTSJ, we redefine the single parent rule as follows:

"The parent of a scoped area is the area in which the object representing the scoped area is allocated."

Then, we propose to base the parentage relationship on the way that scoped areas are created, instead of the order in which scoped areas have been entered by threads such as in RTSJ. In order to do that we suggest take into account the following modifications [7]:

i) The parentage relation of areas implies to maintain only a *scope tree* structure, which is shared by all real-time thread of the application; instead to maintain a scope stack for each real-time thread, as the current edition of RTSJ suggest.

ii) The `ScopedMemory` class contains the `getOuterScope()` method, which allows us to know, for the current task, the memory area which is prior to entering the current area (i.e., its ancestor). This rule was in the former edition of RTSJ , but not in its current edition. Note that in the current RTSJ specification, this method belongs to the `RealTimeThread` class (see Section 2).

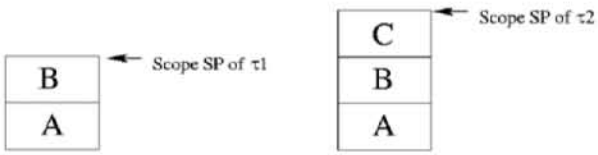
iii) Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count of the number of real-time threads having it as current area (*task-counter*), and also a reference count of the number of scoped areas created within the area (*children-counter*). Note that the current RTSJ specification maintains only a reference counter for real-time threads using the scoped area (i.e., the task-counter). Then, we maintain this reference counter and also we add another reference counter for the children of the memory area. When both task and child reference counters for a scoped memory reach zero, the scoped area is a candidate for reclamation.

3.3 The determinism of our proposed solution

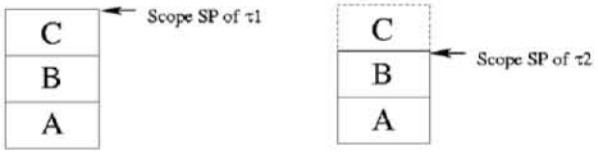
Consider three scoped areas: A, B, and C, which have been created in the following way: the A area has been created within the heap, the B area has been created within the A area and the C area has been created within the B area. That means that the heap was the current area when creating the A object, A was the current area when creating the B object, and B was the current area when

creating the C object. In this way, the creation of the A, B, and C scoped areas gives the following parentage relation: the heap is the parent of A, the area 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 one, whereas for C it is zero.

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. Moreover, task $\tau 2$ has entered areas A and C, which increases by one the task-counter for A and C (see Figure 4.*a*). In this situation, the task-counter for A is two, whereas for B and C is one. If task $\tau 1$ enters the area C and task $\tau 2$ the area B, at different fromthose that occur in RTSJ [11], the single parent rule is not violated. Then, instead of throwing the `ScopedCycleException()`, we have the situation shown in Figure 4.*b*. At this moment, the task-counter for scoped memory areas A, B, and C are two.



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

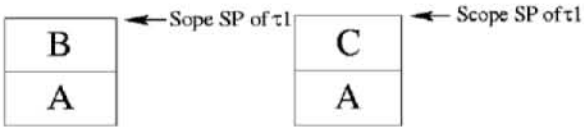


b. $\tau 1$ enters C scoped area and $\tau 2$ enters B.

Figure 4: 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 areas. 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 and creates B and C, which increases its task-counter by one and its child-counter by two, whereas the task-counter and the child-counter of both B and C are zero. Then, task $\tau 1$ enters into scoped areas B (Figure 5.*a*) and C (Figure 5.*b*), which increases by 1 the task-counter of both B and C. In this situation, only references from objects allocated within B or C to objects within A are allowed. Note that it is not possible for task $\tau 1$ 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 5: 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 the root of the scoped tree [2]. 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. As we can show, our proposed implementation of the 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, and *iii)* the parentage relation does not change during the scoped memory life.

3.3 Checking the assignment rules

Since assignment rules cannot be fully enforced by the compiler, some dangling pointers must be detected at runtime [2]. The more basic approach is to take the advice given in the current edition of the RTSJ specification [11]. That is to introduce a code to explore the scope stack associated to the current task, in order to verify that the scoped area from which the reference is created was pushed in the stack before than the area to which the referenced object belongs. This approach requires the introduction of write barriers; that is to take actions in each *store* operation. 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 area 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 [6].

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. The ancestor relation among scoped memory areas is defined by the nesting areas themselves. Since in our proposed implementation, the parentage relation changes at determined moments (i.e., when creating or collecting a scoped area) we can use a name-based technique, which facilitates constant-time checking for the assignment rules. The management of memory areas names only requires to copy the parent name and to include the new created area identifier at the end of it when creating a scoped area, and to invalidate it when the area is collected. Consider three scoped areas: A, B, and C with the following parentage relation: the heap is the parent of A, the area A is the parent of B, and B is the parent of C. Then, the name of the area A is '**A**', the name of area B is '**AB**', and the name of the area C is '**ABC**' (see Figure 6).

Our parentage relation is less dynamic than in the current RTSJ edition, where the parent-child relationship changes as scoped memory areas are entered and exited. In our solution, the parent-child relationship only changes when creating or destroying a scoped memory area (i.e., when the children reference count increases or decreases). Then, the structure of the scope tree is not affected, when entering/exiting a memory area or creating/destroying a thread.

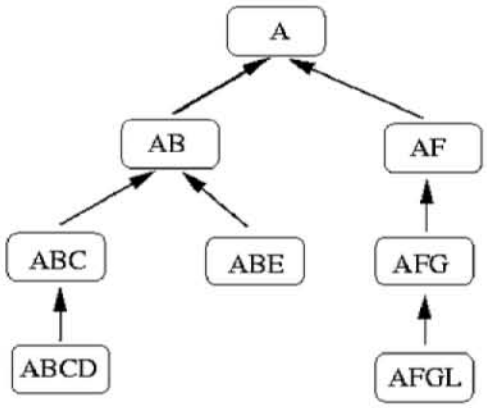


Figure 6: Memory area tree structure.

Figure 7 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 = name of the region to which the x object belongs;
Y = name of the region to which the y object belongs;
if ((Y and X) <> Y)) illegalAssignment();

Figure 7: Checking the assignment rules.

4.3 Estimating the write barrier overhead

We consider that the time cost to detect illegal assignments is a fraction of the total program execution time. Then, to obtain the overhead that write barrier introduces, two measures are combined, the number of events, and the cost of the event. All the objects created in Java are allocated in the heap (i.e., dynamic memory that in RTSJ may be within either the heap or another memory area); only primitive types are allocated in the runtime stack [4].

In most applications of the SPECjvm98 benchmark [12], less than half (i.e., 45%) of the references are to objects within the heap rather than primitive types (e.g., bytes or integers), the other half is to either the *Java or the native stack* (see Table 2). We also notice that about 35% of the total executed bytecodes requires an object reference, where typically 70% is for load operations and 30% for store operations. Then, 15% (i.e., 0.45*0.35) of the bytecodes reference an object within the heap, where 10% (i.e., 0.15*0.30) of the bytecodes requires write barriers avoiding illegal assignments. As a conclusion, 5% i.e., 0.15*0.30) of the executed bytedodes requires write barrier executions.

	Executed Bytecodes	Object Accesses	% Object Accesses	% Heap Accesses
JESS	1,820	707	38.84	39.40
DB	3,700	1,646	38.56	45.61
JAVAC	1,953	724	37.07	28.70
MTRT	2,122	575	27.09	50.97
JACK	2,996	1,022	34.11	50.74

Table 2. Memory reference behavior.

We also use an artificial collector benchmark which is an adaptation made by Hans Boehm from the John Ellis and Kodak benchmark[3]. 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[8].

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 0.8%. Nevertheless, 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 area to which the two objects of the reference belong.

5 Conclusions

The proposed parentage relation of memory areas allow us to use a name-based technique to check illegal references, which simplifies the suggested RTSJ implementation based on a scope stack. Since checks for illegal references requires actions before each assignment statement, which adversely affects both the performance and predictability of the RTSJ application, our suggested parentage relation results particularly interesting.

Our proposed solution requires that every scoped area have two reference counters associated to it. Note that by collecting areas, problems associated with reference-counting collectors are solved: the space and time to maintain two reference-counts per scoped area is minimal, and there are no cyclic scoped area references. Note that the introduction of this change in the parentage relation simplifies the complex semantics for scoped memory areas adopted by RTSJ.

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] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. "Project Golden Gate: Towards Real-Time Java in Space Missions". The 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC). IEEE 2004.

[4] D. Gay and B. Steensgaard. Stack Allocating Objects in Java. Technical report, Research Microsoft, 1998.

[5] M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Java Embedded Real-Time Systems: An Overview of Existing Solutions". In Proc. of the 3th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), pages 392–399. IEEE, March 2000.

[6] 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.

[7] M.T. Higuera-Toledano. "Towards an Understanding of the Behaviour of the Single Parent Rule in the RTSJ Scoped Memory Model". In Proc. Of the 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS). IEEE 2004.

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

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

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

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

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

[13] T. Zhao, J.Noble, and J. Vitek. "Scoped Types for Real-Time Java", In Proc of 25th IEEE International Real-Time Systems Symposium(RTSS) 2004.