Analyzing the Performance of Memory Management in RTSJ

Teresa Higuera, Valérie Issarny

Analyzing the Performance of Memory Management in RTSJ*

M. Teresa Higuera-Toledano and Valérie Issarny
INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cédex, France
Email: Teresa.Higuera@inria.fr

Abstract

The memory models used in the Real-Time Specification for Java (RTSJ) can incur high amounts of overhead; It is possible to reduce this overhead by taking advantages of hardware features. This paper provides an indepth analytical investigation of the overhead of write barriers in RTSJ VMs, and describes and analyzes some solutions to reduce the overhead of write barriers.

`Keywords:` Java, Real-Time, Embedded, Garbage Collection, Memory Regions, Write Barriers, Performance.

1. Introduction

RTSJ distinguishes between three kinds of tasks: *low-priority*, that are tolerant with the Garbage Collector (GC); *high-priority*, that cannot tolerate unbounded preemption latencies; and *critical*, that cannot tolerate preemption latencies. Low-priority tasks are instances of the `Thread` class, high-priority tasks are instances of the `RealtimeThread` class, and critical tasks are instances of the `NoHeapRealtimeThread` class. The `MemoryArea` abstract class supports the Memory Region (MR) paradigm [2] through the three following kinds of regions (see Figure 1): (*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` class) or not (i.e., supported by the `VTMemory` class) on the time taken to create objects; and (*iii*) the conventional heap, supported by the `HeapMemory` class. Objects allocated within immortal MRs live until the end of the application and are never subject to garbage collection. Objects with limited lifetime can be allocated either into a scoped region or the heap. Garbage collection within the heap relies on the (real-time) GC of the JVM. Scoped regions may or may not be subject to internal real-time garbage collection depending on their temporal properties[1]. However, since RTSJ does not impose GC within scoped regions, we consider in this paper that scoped regions are never garbage collected. A scoped region gets collected as a whole once it is no longer used.

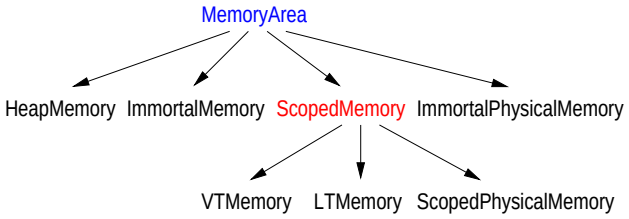

Figure 1. The `MemoryArea` hierarchy in RTSJ.

RTSJ further defines the `GarbageCollector` abstract class, which can be customized through an incremental collector allowing the application to execute while the GC has been launched. In the following we assume the use of the GC algorithm given in [8] (i.e., the *four-color* algorithm that builds on [1]): an object within the heap is colored white when not reached by the GC, black when reached, and grey when it has been reached, but its descendants may not be; and an object outside the heap is colored red. Grey objects make a wavefront, separating the white (unreached) from the black (reached) objects, and the application must preserve the invariant that no black objects have a pointer to a white object, which is achieved using write barriers in [18]. The collection is completed when there are no more grey objects. All the reached objects within the heap are black (those that are reachable from the heap roots and from outside the heap), there is no grey object, and all the white objects can be recycled[2]. Red objects having pointers to object within the heap (i.e., black, grey, or white objects) are considered as *external roots* for the GC.

*This work has been partially funded by Texas Instruments.

[1] We can build a `VTMemory` object with a specific GC. Note that in this case, critical tasks must be able to use it.

[2] The recycling of objects is actually done after finalization.

A thorough analysis of the parameters influencing the performance of the memory management in RTSJ is presented regarding both the management of MRs (Section 2) and real-time GC within the heap (Section 3). We use 5 SPECjvm98 [14] (see Table 1) and an artificial collector benchmark to analyze the behavior of Java applications regarding memory usage. This allows us to have an estimation of memory usage within a memory region. In Section 4, we evaluate the overhead introduced by three different write barrier solutions supporting both MRs and incremental GC. In Section 5, we implement a prototype within the KVM [16] by modifying the original collector to make it incremental, and introducing MRs. Finally, a summary of our contribution concludes this paper (Section 6).

Program	Description
JESS	Expert Shell System based on NASA's CLIPS system.
DB	Emulates data operations on resident memory.
JAVAC	Java compiler from the JDK 1.0.2.
MTRT	Multithreaded raytracer.
JACK	Parser generator (early JavaCC version).

Table 1. Used SPECjvm98 programs.

2. Analyzing the Performance of Regions

In general, the management of memory regions introduces overhead, which we characterize in this Section. The region implementation given in [3] presents an overhead that is constant per instruction executed. RTSJ imposes strict rules on objects access and assignments within regions, the JVM must detect both illegal accesses and assignments and throw an exception when they occur, which introduces high overhead.

2.1. Memory Management Overhead

In RTSJ, each MR supports objects that are related regarding associated lifetime and real-time requirements. Whereas the heap and immortal regions end with the application, a scoped region gets collected by a reference-counting GC once it is no longer used. Then, the overall cost introduced by scoped region management is given by the cost associated with: *(i)* Region creation, which is not considered by RTSJ[3]. *(ii)* Reference counter updates, where we notice that problems associated with reference counting collectors are solved[4]. *(iii)* Object allocation, where the time to allocate an object is proportional to the object size, and in the worst case may include time to acquire additional memory for the region[5]. *(iv)* Region deletion, where before cleaning a scoped region, the root-list of the GC is updated to remove all the objects in the region that are external roots for the GC, and the objects within the terminated region are added to the finalize-list of the GC. *(v)* Checks on objects access/assignment, the efficient of which is discussed in the remainder of this paper.

To support critical applications in RTSJ, the GC of the heap must be disabled and all MRs (i.e., scoped and immortal physical) must be created at initialization time [13]. In this way, the application runs with static memory, which facilitates an accurate pre-runtime analysis.

2.2. Illegal Accesses and Assignments

A reference from a critical task to an object allocated in the heap causes the `MemoryAccessError` exception. *Illegal accesses* must be checked when executing instructions that load references within objects or arrays, e.g., by introducing the following *read barriers* for each load reference:

if ((τ = critical) and (region(Y) = heap)) goto illegalAssignment:;

The lifetime of objects allocated in scoped regions is governed by the control flow: *(i)* objects within either the heap or an immortal region cannot make assignments to objects within a scoped region, and *(ii)* objects within a scoped region cannot make assignments to objects within an non-outer scoped region. *Illegal assignments* causes an `IllegalAssignmentError` exception, and must be checked when executing instructions that store references within objects or arrays, e.g., by introducing the following *write barriers* for each load reference:

```
if (region(Y) = scoped)
   if (region(X) = scoped) nestedRegions(X,Y)
   else goto illegalAssignment:;
```

The `nestedRegions(X,Y)` function is based on a region stack associated with the active task (see Figure 2) and throws the `MemoryAccessError()` exception when the region to which the object X belongs is not found in the region stack, and the `IllegalAssignmentError()` exception when the region to which the object X belongs is not inner to the region to which the object Y belongs.

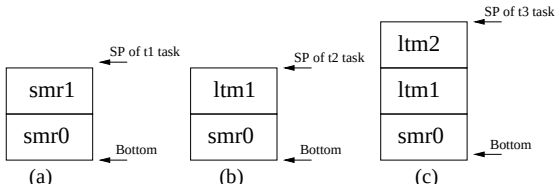

Figure 2. Region stack: *(a)* t_1**,** *(b)* t_2**, and** *(c)* t_3**.**

[3] RTSJ does not consider the execution time of object constructors.

[4] The space to store reference counters is minimal, and there cannot be cycles among regions.

[5] Whereas an allocation in a `VTMemory` region may take variable time, the time taken in a `LTMemory` region is linear to the object size.

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

We consider that the time cost to detect both illegal accesses and assignments is a fraction of the total program execution time. All the objects created in Java are allocated in the heap (i.e., dynamic memory, that in RTSJ may be either within the heap or another MR); only primitive types are allocated in the runtime stack [4]. In most applications of the SPECjvm98 benchmark, 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 [9]). 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., 0.15×0.70) of the bytecodes requires read barrier avoiding illegal accesses of critical tasks to objects within the heap, and 5% (i.e., 0.15×0.30) write barriers avoiding illegal inter-region assignments.

	Executed Bytecodes	Object Accesses	% Object Accesses	% Heap References
JESS	$1,820 \times 10^6$	707×10^6	38.84	39.40
DB	$3,700 \times 10^6$	$1,464 \times 10^6$	39.56	45.61
JAVAC	$1,953 \times 10^6$	724×10^6	37.07	28.70
MTRT	$2,122 \times 10^6$	575×10^6	27.09	50.97
JACK	$2,996 \times 10^6$	$1,022 \times 10^6$	34.11	50.74

Table 2. Memory reference behavior.

We use write barriers to detect illegal accesses[6], as well to maintain the root-set of the GC[7] and to preserve the invariant that no black object references a white one, called *tri-color* invariant [1]. As a conclusion, we have 5% (i.e., 0.15×0.30) as a maximum bound for write barrier executions.

RTSJ does not consider the write barrier overhead for MRs, then we add the `getWriteBarrierOverhead()` method to the `MemoryArea` abstract class, which gives the cost to detect illegal assignments between different types of MRs. In the same way, we add the `getWriteBarrierOverhead(int n)` method to the `ScopedMemory` abstract class, which identifies the write barrier cost to have `n` nested levels for scoped regions.

3. Analyzing the Collection Performance

We can determine the performance of an incremental GC through the following parameters: (*i*) the ratio of the amount of allocated objects with the total size of the heap (*memory utilization*) which relates to the reclamation rate, (*ii*) the space and time needed by the collection (*overhead*), (*iii*) the duration of collection pauses (*latency*), and (*iv*) the effort to coordinate the application and the collector (*write barrier overhead*). Ideally, the memory utilization should be high so that the GC does not run frequently, the overhead should be low to improve the performance of applications, and the latency must be low and bounded for real-time applications. We analyze the aforementioned parameters in the following.

To simplify our presentation, we do not treat fragmentation assuming that all the objects have the same size. In that context, a GC *pass* is hereafter used to mean the overall execution of the GC once it is launched, from the tracing of the object graph to the reclamation of dead objects. A GC *increment* is further used to mean actual GC execution. It is also important to note that in our algorithm, memory that becomes garbage is freed at the end of the GC pass (i.e., new objects are allocated black).

3.1. Reclamation Rate

The collector must terminate before the free memory gets exhausted. A usual strategy to avoid the application to run out of memory is to accelerate the GC according to the application's allocation rate, which can be computed as the $\frac{amount\ of\ dynamic\ memory\ used}{number\ of\ executed\ instructions}$ (see Table 3 [9]).

	Executed Instructions	Allocated Memory (KB)	Allocation Rate ($\frac{KB}{10^3\ inst}$)
JESS	$5,328 \times 10^6$	314,533	60
DB	$9,168 \times 10^6$	99,927	11
JAVAC	$7,717 \times 10^6$	221,206	29
MTRT	$3,917 \times 10^6$	164,444	43
JACK	$6,553 \times 10^6$	207,550	32

Table 3. Allocation behavior.

To ensure the above condition, it is necessary to quantify the worst case allocation rate and to put this measure as a bound. Let L be the maximum amount of live objects, and M be the memory size, we have $M - L$ free memory. Since new objects created during a GC pass will not be collected until the next pass, we must account for this memory occupation (U). We consider that the amount of new objects allocated while tracing, is not greater than the amount of memory used, i.e., $U \leq L$. This implies a minimum safe tracing rate of $\frac{2 \times L}{M - L}$ [18], which approaches zero as memory becomes large[8].

[6] We apply the same optimization as for the incremental GC which is to use write barriers instead of read barriers.

[7] The GC root-set usually includes the local variables in run-time stacks and static variables defined in loaded classes; we must further add objects allocated outside the heap having references to objects within the heap.

[8] Since fragmentation reduces the actual memory available, faster tracing is required.

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

Adapting the reclamation rate. At the end of each pass, the GC can determine how much alive memory has been traced and revise its worst-case, estimating what could be alive at the next pass. When the GC determines that it can reduce the reclamation rate, it may stop its activity and resume later[9]. This improves the performance of the mutator, but not too much since write barriers are still executed when the GC is disabled. Then, it is interesting that our GC supports an efficient way to disable barriers on the fly which has been achieved in [8] by using the picoJava-II hardware support.

3.2. Collection Overhead

The number of times that the GC must be run (N) and the number of instructions executed by a GC pass (I_{GC}), depend on the heap size [9]. The overhead introduced by the GC is inversely proportional to the heap size, and can be given by the following expression: $\frac{N \times I_{GC}}{I}$, where I denotes the total number of instructions executed by the CPU (see Figure 3).

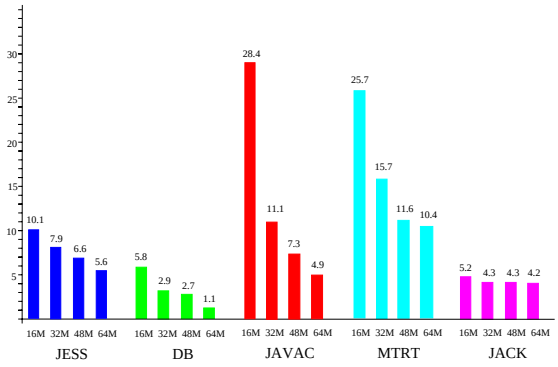

Figure 3. GC overhead.

For an incremental collector, the total effort required to perform a complete GC pass can be configured as a function of the system workload [11]. If S gives the seconds needed by the CPU to complete a collection pass, and G the fraction of CPU dedicated to garbage collection during this time; the time to execute completely the incremental GC is given by $\frac{S}{G}$. Thus, when the GC is executing, the quantity of occupied memory in the heap is $V \times \frac{S}{G}$, where V represents the total bytes allocated per second. Considering new objects created during the GC pass (U), the total memory (M) must be greater than the maximum amount of live objects, i.e., $M > \frac{V \times S}{G} + U$. Then, the minimum fraction of the CPU time spent by the GC is $\frac{V \times S}{M - U}$, which approaches 0 when the amount of memory becomes large or the application allocation rate becomes small.

[9] For instance, if the amount of data is less than $1/3$ of the maximum heap, the GC is disabled (i.e., $L < \frac{1}{3} \times M$ which means $\frac{2 \times L}{M - L} < 1$).

Minimizing the Overhead. An option to minimize the GC overhead is to reduce the number of objects that must be managed by the GC to improve the performance of the GC. Then, improvements on the Java compiler may reduce the GC rate by putting more heap objects in the stack. Some studies show that the percentage of objects that could be allocated in the stack instead of the heap are generally in the $5\% - 15\%$ range, and in some cases as high as 56% [10]. Notice further that objects allocated within immortal and scoped regions of RTSJ are not garbage collected[10], allocating objects in these MRs thus reduces the GC overhead.

3.3. Preemption Latency and Response Time

We analyze here the schedulability of the GC assuming *Rate Monotonic Scheduling* (RMS). Consider N tasks with a priority higher than the one of the GC ($\tau_{GC+1}, .., \tau_{GC+N}$). Each task τ_i has a period T_i and a worst-case execution time C_i. We denote as R_{GC} the worst case response time of the GC, which must be greater than the sum of the worst case execution time of all the tasks with higher priority [5]: $R_{GC} = \sum_{i=GC+1}^{GC+N} (\lceil \frac{R_{GC}}{T_i} \rceil \times C_i)$. To calculate R_{GC} in this recursive formula, we give C_{GC} as the first value for R_{GC}. Then, $S > (R_{GC} + L_{GC}) \times N_{GC}$, where S gives the seconds needed by the CPU to complete a collection pass, L_{GC} the maximum execution time of a collector increment, and N_{GC} the number of increments for each collection pass. The maximum preemption latency that the collector can introduce in the system is $R_{GC} - \frac{S}{N_{GC}}$.

Minimizing Allocation Latency. The fact that the overhead of memory allocation depends on both the size of the object and the heap evolution, makes it intolerable for critical tasks. In order to eliminate this imprecise cost, critical tasks do not execute actions related to the GC, and are never affected by the preemption latency of the GC[11]. Regarding the size of the object, every allocation must have an execution time cost, that is bounded by a linear function on the size of the object; we do not include in this cost static initializations associated with the object nor the execution of its constructor. For this purpose, RTSJ defines the `LTMemory` class (a `ScopedMemory` subclass) that represents a memory area guaranteed by the system to have constant time allocation. This class allows tasks to allocate objects, ignoring reclamation and avoiding delays because of the GC. Thus, it is safe to associate a `LTMemory` object with critical tasks.

[10] This is always the case for objects within an instance of `LTMemory` but this is not mandatory for objects within an instance of `VTMemory`.

[11] RTSJ critical tasks are not allowed to allocate or even reference objects from the Java heap.

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

3.4. Write Barrier Overhead

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

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

Note that the execution time taken by the `greyObject(Y)` function is considered as part of the GC overhead rather than as part of the write barrier overhead. The `GarbageCollector` abstract class of RTSJ does not support the `getWriteBarrierOverhead()` method[12]. Since the heap coexists with other MRs, we consider that this method must also be implemented for all collectors to give the overhead caused by detecting illegal assignments of critical task to objects within the heap. For *mark-and-sweep* collectors, this method further gives the overhead caused by the write barriers introduced to detect when to update the collector's root-set:

```
if ((color(X) = red) and (color(Y) ≠ red)) updateRootSet(X, Y);
```

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

4. Evaluating the Write Barrier Cost

In this Section, we first propose three different write barrier implementations to support the RTSJ memory model. Next, we estimate the write barrier overhead introduced by both the collector and memory regions in the proposed solutions.

[12]In RTSJ, the `getWriteBarrierOverhead()` method is supported by the `IncrementalGarbageCollector` class.

4.1. Write Barrier Implementations

Solution 1. ***Modifying the Java Interpreter.*** This solution consists in modifying the JVM by introducing the code given in Figure 4 in the interpretation of each bytecode whose function consists in assigning an object Y to another object X[13].

```
if (region(Y) = scoped)
    if (region(X) = scoped) nestedRegions(X,Y)
    else goto illegalAssignment:;
if ((τ = critical) and (region(Y) = heap)) goto illegalAssignment:;
if ((color(X) = red) and (color(Y) ≠ red)) updateRootSet(X, Y)
else if ((color(X) = black) and (color(Y) = white)) greyObject(Y);
```

Figure 4. Write barrier code.

Solution 2. ***Using Existing Hardware.*** We improve the performance of Solution 1 by using the write barrier support of the picoJava-II microprocessor, as proposed in [8]. In this solution, write barriers must be configured at context-switch time depending on the scheduled task. Non-critical tasks throw the `gc_notify` exception when a white object is assigned to a black one, or when an object is assigned to another one allocated in a different MR. Whereas critical tasks throw the `gc_notify` exception when the assigned object is within the heap, or a different MR that the other one. both objects are allocated in different MRs. The code executed by the `gc_notify` exception handler is the same as the one introduced in the interpreter in the former solution (see Figure 5).

```
gc_notify:
    if (region(Y) = scoped)
        if (region(X) = scoped) nestedRegions(X,Y)
        else goto illegalAssignment:;
    if ((τ = critical) and (region(Y) = heap)) goto illegalAssignment:;
    if ((color(X) = red) and (color(Y) ≠ red)) updateRootSet(X, Y)
    else if ((color(X) = black) and (color(Y) = white)) greyObject(Y);
    priv_ret_from_trap;
```

Figure 5. Handling the `gc_notify` exception.

Solution 3. ***Modifying the Existing Hardware.*** This solution modifies the hardware support of picoJava-II to have three different traps (see Figure 6). In this solution, non-critical tasks cause the execution of: *(i)* the `gc_notify_1_0`

[13]The bytecodes causing write barriers are: `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, and `aastore_quick`.

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

exception when a non-red object is assigned to a red one, *(ii)* the `gc_notify_1` exception when any object is assigned to another one allocated in a different MR, and *(iii)* the `gc_notify_0` exception when a white object is assigned to a black one. Critical tasks cause also the `gc_notify_0` exception when a non-red object is assigned.

```
gc_notify_1_0:
    if ( τ ≠ critical) updateRootSet(X,Y) else goto illegalAssignment:;
    priv_ret_from_trap
gc_notify_1:
    if (region(Y) = scoped) nestedRegions(X,Y);
    priv_ret_from_trap
gc_notify_0:
    if ( τ ≠ critical) greyObject(Y) else goto illegalAssignment:;
    priv_ret_from_trap
```

Figure 6. Write barrier exception handlers.

4.2. Evaluating the Write Barrier Overhead

Axioms and Theorem. We are interested in fixing a maximum bound for the number of events that: *(i)* makes an inter-region assignment, *(ii)* explores the region stack, *(iii)* creates an external reference for the collector, and *(iv)* attempts to break the tri-color invariant. We assume here that each object has an equal probability to being referenced.

Notations:

Let r, b, g, and w, be respectively the number of red, black, grey, and white objects, and h, i, and s be respectively the number of objects within the heap, an immortal region, or a scoped region, found in the system at a given instant. Let further, x and z denote respectively the number of inter-region and intra-region assignments, found in m assignments made by the task τ.

Axiom 1. $\frac{h}{x} + \frac{i}{x} + \frac{s}{x} = 1$

In x inter-region assignments of the task τ, there are h assignments from thee heap, i assignments from an immortal region, and s assignments from a scoped region.

Axiom 2. $\frac{b}{h} + \frac{g}{h} + \frac{w}{h} = 1$

In h objects within the heap there are b objects black, g objects grey, and w objects white.

Theorem:

The probability that a task τ breaks the tri-color invariant when making m assignments is bounded by $0.25 \times h$.

Proof.

We have $h = b + g + w$. We can further express the probability to break the tri-color invariant as $\frac{b \times w}{h^2} = \frac{b \times (h-(b+g))}{h^2}$, this probability is maximum when there are no grey objects in the system (i.e., $h = b + w$). Then: $b \times (h - (b + g)) <= b \times h - b^2$. Where the $b \times h - b^2$ expression takes its maximum value for $b = \frac{h}{2}$ (i.e., $0 = h - 2 \times b$) and $w = \frac{h}{2}$ (i.e., $h = \frac{h}{2} + w$).

Quantifying the Overhead. To obtain the write barrier overhead solutions given in § 4.1, two measures are combined: (*i*) the number of events (E), and (*ii*) the measured cost of the event (C). We also take into account the percentage of bytecodes requiring write barriers, which has been evaluated as 5% in § 2.2. Then, we compute the total write barrier overhead introduced by both MRs and the GC:

$$MR_{Ov} = 0.05 \times (E_{MR} \times C_{MR} + E_{scoped} \times C_{scoped})$$
$$GC_{Ov} = 0.05 \times (E_{GC} \times C_{GC} + E_{incGC} \times C_{incGC})$$

Where C_{MR}, C_{scoped}, C_{GC}, and C_{incGG} parameters correspond to:

$$C_{MR} = \text{MemoryArea.getWriteBarrierOverhead()}$$
$$C_{scoped} = \text{ScopedMemory.getWriteBarrierOverhead(n)}$$
$$C_{GC} = \text{GarbageCollector.getWriteBarrierOverhead()}$$
$$C_{incGC} = \text{IncrementalGC.getWriteBarrierOverhead()}$$

Event parameters. We then estimate the maximum probability to execute the write barrier code when a non-critical task makes an assignment, as given in Table 4. Note that for critical tasks, the overhead due to the GC is 0 (i.e., E_{GC} and E_{incGC} equal to zero, otherwise the `IllegalAssignmentError()` exception raises).

Events	Solution 1	Solution 2	Solution 3
E_{MR}	1	$\frac{x}{m} + 0.25\frac{h}{m}$	$\frac{x}{m} - \frac{h}{m}$
E_{scoped}	$\frac{s}{m}$	$(\frac{x}{m} + 0.25\frac{h}{m})(\frac{s}{m})$	$\frac{s}{m}$
E_{GC}	1	$\frac{x}{m} + 0.25\frac{h}{m}$	$\frac{h}{m}$
E_{incGC}	$1 - \frac{x}{m}$	$(\frac{x}{m} + 0.25\frac{h}{m})(1 - \frac{x}{m})$	$0.25\frac{h}{m}$

Table 4. Max bound on write barrier events.

Cost parameters. The write barrier cost is proportional to of the number of evaluated conditions. Then, we bound the cost parameters as $maxConditions \times \frac{conditionCost}{assignmentCost}$. Where the $maxConditions$ parameter is the maximum number of evaluated conditions to check whether the following actions should be executed: *(i)* call `nestedRegions(X, Y)`, *(ii)* execute `nestedRegions(X, Y)`, *(iii)* call `updateRootSet(X, Y)`, and *(iv)* call `greyObject(Y)`. And the $conditionCost$ parameter is the execution time to evaluate a condition. Table 5 gives the maximum and average value for the number of evaluated conditions, where n is the maximum number of nested scoped levels.

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

Cost Parameter	Solution 1 and 2		Solution 3	
	Maximum	Average	Maximum	Average
C_{MR}	2	1	1	0.5
C_{scoped}	n	$\frac{n}{2}$	n	$\frac{n}{2}$
C_{GC}	3	1.5	1	0.5
C_{incGC}	2	1	1	0.5

Table 5. Evaluated conditions for write barrier.

Bounding the Overhead. Let MR_{Ov_i} and GC_{Ov_i} $(1 <= i <= 3)$ be the MR_{Ov} and GC_{Ov} parameters of each given solution, then:

$$MR_{Ov_1} < 0.05(2 + n\frac{s}{m}) \times \frac{conditionCost}{assignmentCost}$$
$$MR_{Ov_2} < (\frac{x}{m} + 0.25\frac{h}{m}) \times MR_{Ov_1}$$
$$MR_{Ov_3} < 0.05(\frac{i}{m} + (n+1)\frac{s}{m}) \times \frac{conditionCost}{assignmentCost}$$

$$GC_{Ov_1} < 0.05(3 + 2(1 - \frac{x}{m})) \times \frac{conditionCost}{assignmentCost}.$$
$$GC_{Ov_2} < (\frac{x}{m} + 0.25\frac{h}{m}) \times GC_{Ov_1}$$
$$GC_{Ov_3} < 0.6\frac{h}{m} \times \frac{conditionCost}{assignmentCost}$$

Note that for hardware-based solutions (i.e., solutions 2 and 3) we must take into account the time that the picoJava-II microprocessor spends to catch a trap. Recall also that the write barrier overhead introduced by scoped regions is the execution time of the `nestedRegions(X,Y)` function. Then, to bound it, we must bound the number of nested region levels.

Comparison. In solution 1, the write barrier code is executed for both inter-region and intra-region references. Solution 2 reduces the cost of write barriers for intra-region references to the cost to maintain the tri-color invariant (i.e., by a factor of $\frac{x}{m}+0.25\frac{h}{m}$). This is because the `gc_notify` exception traps only when a task makes an inter-region reference or attempts to violate the tri-color invariant. Solution 3 minimizes the cost for inter-region references, to the cost to detect both illegal assignments when the referenced object is outside the heap and root-set updates when the referenced object is within the heap.

5. Experiment

We have modified the KVM garbage collector[14] making it a stack-based tri-color algorithm. We have implemented the `IncrementalGC` class within the KVM by modifying some files[15]. This class supports the method related with parameters characterizing the collector behavior: `getMinimumReclamationRate()`, `setReclamationRate()`, `getOverhead()`, `getWriteBarrierOverhead()`, and `getPreemptionLatency()`. We have only implemented three types of memory regions: the heap that is collected by an incremental GC, immortal that are never collected and can not be nested, and scoped that have limited live-time and can be nested. These regions are supported by the `HeapMemory`, the `ImmortalMemory`, and the `ScopedMemory` classes. Unlike RTSJ, in our prototype the `ScopedMemory` class is a non-abstract class, and the `MemoryArea` abstract class has not been implemented[16]. The `getWriteBarrierOverhead()` method has been implemented for the three classes.

Instead of using the SPECjvm98 benchmark, which is not compatible with the KVM, we use an artificial collector benchmark. This is an adaptation made by Hans Boehm from the John Ellis and Pete Kovac benchmark[17]. Two data structures of the same size are kept around during the entire process: *(i)* a tree containing many pointers and *(ii)* a large array containing double precision floating point numbers, which we have modified to contain integers to make it compatible with the KVM. This benchmark executes 262×10^6 bytecodes and allocates $408\ MBytes$. Then, the allocation rate is about $1.6\ KBytes/1000\ executed\ bytecodes$. The number of garbage collection pass, the microseconds spent in garbage collection, and the percentage overhead introduced by our collector are given in Table 6:

Memory Heap	GC pass	Collecting Time	Execution Time	% Overhead
$8MB$	51	13.54×10^6	72.87×10^6	18.85%
$16MB$	27	13.17×10^6	72.72×10^6	18.11%
$24MB$	17	12.80×10^6	71.99×10^6	17.80%
$32MB$	13	11.82×10^6	70.50×10^6	16.50%

Table 6. Garbage collection overhead.

The maximum latency to preempt the incremental collector has been measured as $1\mu second$. The number of executed bytecodes performing write barrier test is 15×10^6 (i.e., `aastore:` 1×10^6, `putfield:` 6×10^6, `putfield_fast:` 7×10^6, `putstatic:` 19, and `putstatic_fast:` 0) for a total of 262×10^6 executed bytecodes. This means that 5% of executed bytecodes perform a write barrier test, as already obtained in § 2.2 with SPECjvm98 [14]. And the overhead introduced by the software write barrier test in each assignment, is:

- 45% to maintain the root-set.
- 31% to preserve the tri-color invariant.

[14] Version 1.0.1

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

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

[17] $http://www.hpl.hp.com/personal/Hans\_Boehm/gc/gc\_bench.html$

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

- 31% to detect illegal references.
- 16% to check a nested scoped level[18].

6. Conclusion

A real-time GC avoids the user to recycle memory, but introduces high overhead and unpredictable behavior. Memory regions, which can be supported in a stack discipline offer a high level of predictability. The memory regions model of RTSJ combines the advantages of both techniques. But, this model introduces high write barrier overhead, and it is not clear that this approach is better than classical static memory, classical real-time collection, or classical memory regions. The contribution of our work comes from the adaptation and integration of relevant solutions to make memory reclamation real-time, in the context of RTSJ, and is based on the analysis of the parameters that are the most influential in memory management performance.

In this paper, we have analyzed and estimated the performance of the RTSJ memory model. To this end, we have studied the memory behavior of the SPECjvm98 applications. These non-real-time applications allocate all object references (i.e., non-primitive types) within the JVM heap (i.e., do not use any other memory region), and do not impose to the collector real-time restrictions. However, we obtain, as a as conclusion, that 5% of the executed bytecodes makes an assignment of an object within dynamic memory. We extrapolate this result to RTSJ, concluding that 5% of the bytecodes executes write barriers to detect *(i)* illegal accesses and assignment introduced by MRs, *(ii)* external roots for the GC, and *(ii)* violations of the tri-color invariant introduced by an incremental GC.

Our solution, for improving performance of memory management partly addresses the use of hardware aid by exploiting existing hardware support for Java (i.e., picoJava-II). A detailed analysis of three different implementations of write barrier shows that the hardware aid improves highly the application performance. Finally, we have integrated our real-time GC and support for memory regions within the KVM, which we have evaluated using and an artificial benchmark designed to analyze the memory behavior. For this prototype we obtain the same proportion of bytecodes requiring write barriers (i.e., 5%).

References

[1] E. Dijkstra, L. Lamport, A. Martin, and C. S. E. Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

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

[3] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proc. of the Conference of Programming Language Design and Implementation (PLDI)*, pages 313–323. ACM SIGPLAN, June 1998.

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

[5] R. Henriksson. Predictable Automatic Memory Management for Embedded Systems. In *Proc. of the Workshop on Garbage Collection and Memory Management*. ACM SIGPLAN-SIGACT, October 1997.

[6] M. Higuera, V. Issarny, M. Banâtre, G. Cabillic, J. 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.

[7] M. Higuera, V. Issarny, M. Banâtre, G. Cabillic, J. Lesot, and F. Parain. Memory management for real-time java: an efficient solution using hardware support. *Real-Time Systems journal*, 2001.

[8] M. Higuera, V. Issarny, M. Banâtre, G. Cabillic, J. 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, May 2001.

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

[10] C. McDowell. Reducing Garbage in Java. Technical report, http://www.cs.Berkeley.edu, 1997.

[11] K. Nilsen. Adding Real-Time Capabilities to Java. *Communications of the ACM*, 41(6):49–56, June 1998.

[12] A. Petit-Bianco and T. Tromey. Garbage Collection for Java in Embedded Systems. In *Proc. of IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, pages 59–67, December 1998.

[13] P. Puschner and A. Wellings. A Profile for High-Integrity Real-Time Java Programs. In *Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, May 2001.

[14] Standard Performance Evaluation Council. SPEC JVM98 benchmarks. Technical report, 1998. http://www.spec.org/osg/jvm98.

[15] Sun Microsystems. picoJava-II Programmer's Reference Manual. Technical report, http://www.sun.com/microelectronics/picoJava, March 1999.

[16] Sun Microsystems. KVM Technical Specification. Technical report, Java Community Process, May 2000.

[17] The Real-Time for Java Expert Group. Real-Time Specification for Java. Technical report, RTJEG, June 2000. http://www.rtj.org.

[18] P. Wilson. Uniprocessor Garbage Collection Techniques. Technical report, ftp://ftp.cs.utexas.edu/pub/garbage/bigsur.ps, 1994.

[18] We have limited to 32 the number of region the number of scoped nexted level to 6, which allows us to support the region stack in a word.

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE