CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Improving the Memory Management Performance of *RTSJ*

M. Teresa Higuera-Toledano[1]
Valérie Issarny[2] [*,†]

[1] *Universidad Complutense de Madrid, Ciudad Universitaria, Madrid 28040, Spain*
[2] INRIA-*Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cédex, France*

SUMMARY

From a real-time perspective, the Garbage Collector (GC)introduces unpredictable pauses that are not tolerated by real-time tasks. Real-time collectors eliminate this problem but introduce a high overhead. Another approach is to use Memory Regions (MR) within which allocation and deallocation is customized. This facility is supported by the memory model of the Real-Time Specification for Java (RTSJ). RTSJ imposes strict access and assignment rules to avoid both, dangling inter-region references and delays of critical tasks because the GC. A dynamic check solution can incur high amounts of overhead, which it is possible to reduce by taking advantages of hardware features. This paper provides an indepth analytical investigation of the overhead introduced by dynamic assignments checks in RTSJ, describing and analyzing several solutions to reduce the introduced overhead.

KEY WORDS: *Java, Real-time, Embedded Systems, Garbage Collection, Memory Regions, Java Microprocessor, Write Barriers, Performance.*

1. INTRODUCTION

The original Java platform provides attributes that make it a powerful platform to develop embedded real-time applications, however, presents some important lacks regarding its use in REAL-TIME systems. The *N*ational Institute of Standards and Technology (NIST), has produced a basic requirements document for a standard real-time Java API extension.

---

[*] Correspondence to: M. Teresa Higuera-Toledano, DACYA, Facultad de Informática, Universidad Complutense de Madrid, Ciudad Universitaria, Madrid 28040, Spain

[†] E-mail: mthiguer@dacya.ucm.es

Contract/grant sponsor: Ministry of Education of Spain (CICYT); contract/grant number: TIC2002–00334

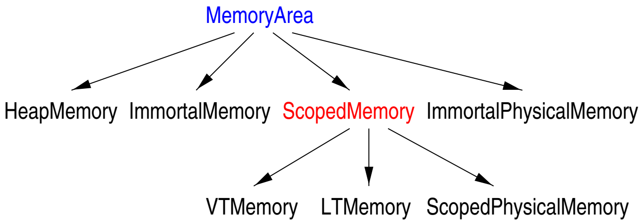

Figure 1. The `MemoryArea` hierarchy in RTSJ.

Solutions that complies with this document are the *R*eal-time Specification for Java (RTSJ) [8] and the *Real-time Core Extension for the Java Platform* [2]. We have analyzed how these solutions resolve the problems that Java presents to effectively support embedded real-time applications [4]. As a conclusion to our study, we found that the RTSJ is the most adequate solution for a Java environment aimed at embedded systems executing multimedia applications (e.g., wireless PDAs), even if some issues are still open (e.g., memory management).

This paper focuses on to make Java memory management real-time while accounting for relevant Java specifications: the RTSJ, the KVM [5] targeting limited-resource and network connected devices, and the microprocessor core *picoJava* [6].

1.1. Memory Areas in RTSJ

The `MemoryArea` abstract class supports the region paradigm in the RTSJ specification [7] through three kinds of regions (see Figure 1): (*i*) immortal memory, supported by the `ImmortalMemory` and the `ImmortalPhysicalMemory` classes, that contain 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. An application can allocate memory into the system heap, the immortal system memory region, several scoped memory regions, and several immortal regions associated with physical characteristics. Objects allocated within immortal MRs live until the end of the application and are never subject to garbage collection. Objects with limited lifetime can be allocated into the heap or a scoped region. Garbage collection within the application heap relies on the (real-time) GC of the JVM. 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.

RTSJ makes distinction between three main kinds of tasks: *(i) low-priority tasks* that are tolerant with the GC, *(ii) high-priority tasks* that cannot tolerate unbounded preemption latencies, and *(iii) critical tasks* that cannot tolerate preemption latencies. Whereas high-priority tasks require a real-time GC, critical tasks must not be affected by the GC, and as

Table I. Assignment rules in RTSJ.

	Reference to Heap	**Reference to Immortal**	**Reference to Scoped**
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	same or outer

a consequence cannot access any object whithin the heap. A scoped region gets collected as a whole once it is no longer used. Then, since immortal and scoped MRs are not garbage collected, they may be exploited by critical tasks, specially `LTMemory` objects, which guarantee allocation time proportional to the object size. The lifetime of objects allocated in scoped regions is governed by the control flow. Strict assignment rules placed on assignments to or from MRs prevent the creation of dangling pointers (see Table I). Several related threads, possibly real-time, can share a MR, and the region must be active until at least the last thread has exited. The strict assignment rules imposed by RTSJ to avoid dangling inter-region pointers incur high amounts of overhead, which it is possible to reduce by taking advantages of hardware features.

1.2. Related Work

The JVM must check for the above assignment rules before to execute an assignment statement, and throw an `illegalAssignment()` exception, if they are violated. This check includes the possibility of static analysis of the application logic [8]. The Tofte-Talpin calculus [21] uses a lexically scoped expression to delimit the lifetime of a region. Memory for the region is allocated when the control enters into the scope of the region constructor, and is deallocated when the control leaves the scope. This mechanism is implemented by a stack of regions where regions are ordered by lifetimes. The allocation and de-allocation of regions is determined at compile time by a *t*ype-based analysis, which consists of to annotate in the source program every expression creating a value with a region variable, where region allocation and de-allocation are explicit. As this solution, our solution is based on a stack of scoped regions ordered by life-times [20]. But given that in RTSJ, a region can be shared among several threads, this solution requires more complex mechanisms because the region will remain active until the last thread has exited, and this fact makes difficult to determine the de-allocation of regions at compile time.

Our proposed solution consists to check the imposed assignment rules preserving dangling pointers dynamically, just when executing the assignment statement. In order to do that, we introduce an extra code in all bytecodes causing an object assignment [18]. This extra code, normally called write barrier must be executed before to update the object reference. A similar approach is given in [22], which uses also a stack-based memory management that operates dynamically. This solution proposes a contaminated GC based on the idea that each object in the heap is alive due to references that begin in the runtime stack. But, at different of our solution, this solution collects memory within the heap, and does not treat another memory region.

As the GC coexists with MRs, objects within the heap having references from objects outside the heap must be considered as roots by tracing-based collectors (i.e., based on copying or mark-and-sweep techniques). In our solution, to maintain the root-set of the GC, we use write barriers. As we use an incremental mark-and-sweep GC strategy based on the tri-colour algorithm given in [10], we need also write barriers to maintain the tri-colour invariant. Hence, to detect new roots of the GC, instead to employ a technique based on the region to which the objects belong (e.g., as used by generational collectors to maintain intergenerational pointers [11]), we introduce a forth colour indicating whether an object is outside the heap [14].

Another problem is the critical tasks that can-not access objects within the heap. In order to detect violations of this rule and thrown a `memoryAccessError()` exception, we use read barriers (i.e., we introduce an extra code in all bytecodes causing an object access). The most common approach to implement read/write barriers is by inline code, consisting in generating the instructions executing write barrier events for every load/store operation. This solution requires compiler cooperation and presents a serious drawback because it increases the size of the application object code [26]. Alternatively, our solution instruments the bytecode interpreter, avoiding space problems, but this still requires a complementary solution to handle native code. A solution minimizing the introduced overhead consists to improve the write barrier performance by using hardware support such as the picoJava-II microprocessor [28], which allows performing write barrier checks in parallel with the store operation. The use of the hardware support for write barriers has been the subject of [18], where we propose to improve the performance of checking *illegal references* by two different ways: using existing hardware support and modifying existing hardware. The performance improvement introduced by these solutions has been compared in [19].

1.3. Paper Organization

In the following, we proposes several memory management solutions more or less compliant with the RTSJ `javax.realtime` package. We first present a software-based solution as a possible implementation of the memory region abstraction presented by the RTSJ (Section 2). Two hardware-based solutions improving the performance of both MR and the GC are then given (Section 3). In Section 4, we evaluate the overhead introduced by three different solutions supporting both MRs and incremental GC. In Section 5, we give some details in order to implement a prototype within the KVM [5], by modifying the original collector to make it incremental, and introducing MRs. Finally, a summary of our contribution concludes this paper (Section 6).

Table II. Used SPECjvm98 programs.

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).

2. THE BASIC STRATEGY

In this section, we introduce a software-based solution to check both illegal assignments and memory access from critical tasks to objects within the heap. Since the RTSJ specification allows the implementation of real-time compliant memory management without prescribing any specific solution, we combine an incremental GC within the heap and a stack-based algorithm supporting the scoped MRs.

2.1. Illegal Assignments

A MR implementation must ensure that the following conditions are checked before the assignment is executed: *(i)* objects within the heap region cannot reference objects within a scoped region, *(ii)* objects within the immortal region cannot reference objects within a scoped region, and *(iii)* objects in a scoped region cannot reference objects within another scoped region that is non-outer. In order to detect *illegal assignments* every real-time thread has associated a region-stack containing all scoped MRs which the thread can hold. The basic idea to detect illegal assignments is to take actions upon those instructions that cause one object to reference another (i.e., by using write barriers):

- The `putfield` (`aputfield_quick`) bytecode causes a reference from an object X to another one Y, and the `aastore` (`aastore_quick`) bytecode stores a reference (Y) into an array of references (X).
- The `putstatic` (`aputstatic_quick`) bytecode causes a reference from an object within persistent memory (i.e., an outermost region) to another object Y.

Then, the MR to which the object Y belongs must be outside of the MR to which the object X belongs. Figure 2 shows the write barrier pseudo-code, that we must introduce in the interpretation of the aforementioned bytecodes, where we denote as X the object that makes the reference, and as Y the referenced object. The region to which an object belongs must be specified in the header of the object. Then, when an object/array is created by executing the `new` (`new_quick`) or `newarray` (`newarray_quick`) bytecode, it is associated with the scope of the active region. The `nestedRegions(X, Y)` function, throws the `IllegalAssignment()` exception when the region to which the Y object belongs is not outer to the region to which the X

```
if ((region(X) $\neq$ scoped) and (region(Y) = scoped)) illegalAssignment();
if ((region(X) = scoped) and (region(Y) = scoped)) nestedRegions(X, Y);
```

Figure 2. Write barrier to detect illegal assignment.

object belongs and returns *true* when the region to which the Y object belongs is outer or equal to the region to which the X object belongs. Following we describe the algorithm of the `nestedRegions(X, Y)` function, which requires two steps:

1. The region-stack of the active task is explored, from the top to the bottom, to find the MR to which the X object belongs. If it is not found, this is notified by throwing a `MemoryAccessError()` exception[†].
2. The region-stack is again explored, but this time we take the MR found in the previous step as the top of the stack. Then, we start the search from the region to which the X object belongs, and the objective is to find the MR to which the Y object belongs (i.e., the region to which the object Y belongs must be outer to the region to which the object X belongs). If the scoped region of Y is not found in the new region-stack, this is notified by throwing an `IllegalAssignment()` exception. If it is found, the `nestedRegions(X, Y)` returns `true`.

2.2. Memory Access Errors

A reference of a critical task to an object allocated in the heap causes the `MemoryAccessError()` exception, which can be achieved by using *read barriers*. Note that read barriers occur upon all object accesses, which means upon executing both types of bytecodes:

- Those causing a *load reference* (i.e., `getfield`, `getstatic`, `agetfield_quick`, `agetstatic_quick`, or `aaload` bytecodes).
- Those causing a *store reference* (i.e., those causing write barriers: `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, or `aastore_quick` bytecodes).

Whereas for bytecodes causing a load reference, we introduce the code given in Figure 3; for bytecodes causing a store reference, we introduce the code given in Figure 4. Where the `type()` function returns `thread`, `task`, and `critical` depending on the type of the parameter task, τ is the active task.

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

```
if ((type(τ) = critical) and (region(X) = heap)) memoryAccessError();
```

Figure 3. Read barrier code for load references.

```
if ((type(τ) = critical) and ((region(X) = heap) or (region(Y) = heap))) memoryAccessError();
```

Figure 4. Read barrier code for store references.

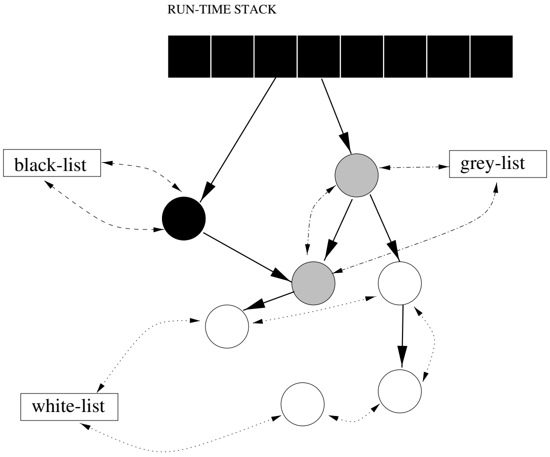

Figure 5. The GC strategy.

2.3. Collecting the Heap

We specifically consider a *mark-and-sweep* GC based on the tri-colour algorithm [10]. The basic algorithm is as follows: an object is coloured white when not reached by the GC, black when reached, and grey when it has been reached, but its descendants may not have been (i.e., they are white). Grey objects make a wavefront, separating the white (unreached) from the black (reached) objects (see Figure 5), and the application must preserve the invariant that no black objects have a pointer to a white object, which is achieved by using write barriers [11].

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

Figure 6. Write Barrier to maintain the tri-colour invariant.

For each thread, we maintain a stack of root pointers. We start the *marking* phase by colouring all objects referenced by root pointers grey. Each root stack pointer is processed root by root, and each object referenced by a root is inserted in a grey-list. If during this phase, the application tries to make a reference from a black object to a white one, the colour of the referenced object is turned grey and the object is moved from the white-list to the grey–list. When all the descendants of a grey object are processed (i.e., the grey object has no white descendant), the grey object is turned black and moved from the grey-list to the black-list. The collection is completed when there are no more grey objects. During the *sweeping* phase, all the white objects can be recycled and all the black objects become white. In this process, objects that must execute the `finalize()` method are moved to the finalize-list. The finalizes are executed by a specialized thread such as in [12]. Finally, for white objects that have finalized, their memory is freed.

The code checking a reference violating the tri-colour invariant (i.e., from a black object to a white one) must be executed when updating an object reference (i.e., when executing the `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, or `aastore_quick` bytecode). Then, we introduce in the interpretation of these bytecodes, the write barrier pseudo-code shown in Figure 6, where the `colour()` function gives the colour of the object parameter, and the `greyObject(Y)` procedure unlinks the Y object from the white-list linking it to the grey-list.

2.4. The GC and MR Interaction

Since objects allocated within regions may contain references to objects within the heap, the GC must take into account these external references, adding them to its reachability graph. To facilitate this task, we introduce a fourth colour (e.g., *red*) meaning that the object is allocated outside the heap (See Figure 7). Hence, we introduce a new invariant:

Definition. **Fourth-colour invariant:** *there are no red objects within the heap, and there are no black, grey or white objects outside the heap.*

The fourth colour allows us to detect when the X object must be added to the root-set of the collector, where the root-list is updated a similar way to how generational collectors maintain *inter-generational pointers* (i.e., by using write barriers).

Note that when using a write barrier-based GC, read barriers detecting memory access errors are not strictly necessary because read operations do not change the colour of the object, then do not affect the GC coherence [14]. The restriction on critical tasks can be reduced to write barriers checks since reads does not interfere with the GC. We apply the same optimization as

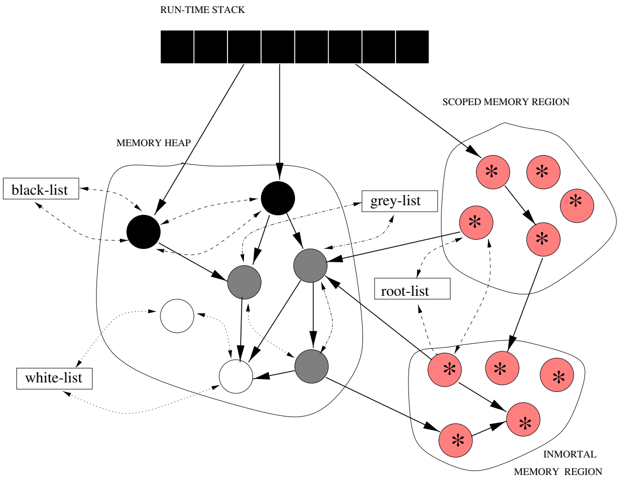

Figure 7. Objects outside the heap are allocated red.

for the incremental GC which is to use write barriers instead of read barriers. In this case, the `MemoryAccessError()` exception which is raised when a critical task attempts to access an object X within the heap, is changed by the `IllegalAssignment()` exception which is raised when a critical task attempts to create a reference to an object Y which belongs to the heap. Whereas the former solution, which is based in both write and read barriers, is 100% compliant with the abstract RTSJ specification; the introduced optimization, which consist to use write barriers in order to detect memory access errors instead to use read barriers, reduces the compliant degree of our proposed implementation solution with RTSJ.

Whereas for non-critical tasks, a reference from a red object (X) to another object (Y) allocated within the heap (i.e., non-red) causes the addition of the X object to the root-set of the collector; for critical tasks, a reference to a non-red object (i.e., white, black, or grey) causes an `IllegalAssignment()` exception. Then we introduce the code of Figure 8, where the `updateRootSet(X, Y)` procedure links the X object to the root-list greying the Y object whether it is white. When the collector explores an object outside the heap (i.e., a root), which has lost its references into the heap, it is eliminated from the root-set. When a scoped MR is freed, all objects within the region having references to the objects within the heap are removed from the root-list of the collector.

As in our solution, objects within the heap are not reallocated, i.e., we use a non-moving collector (no-copying-based and no-having compactation), read barriers can be avoided. In this case, we change write barriers detecting memory access errors for write barriers detecting illegal assignments from critical tasks to objects within the heap. This modification is not compliant with the RTSJ specification, but introduces an important improvement: about 6%

```
if (colour(Y) $\neq$ red)
  if (type($\tau$) = critical) illegalAssignment()
  else if (colour(X) = red) updateRootSet(X, Y)
    else if ((colour(X) = black) and (colour(Y) = white)) greyObject(Y);
```

Figure 8. Write barrier using the red colour.

of the introduced overhead (i.e., whereas the 11% of executed bytecodes performs a load operation, the 5% performs a store operation 4.2).

3. IMPROVING THE PERFORMANCE

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 [26]. 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 [28], which allows performing write barrier checks in parallel with the store operation. This alternative solution has been the subject of [18].

3.1. Using the Write Barrier Support of the picoJava-II Microprocessor

Upon each instruction execution, the picoJava-II core checks for conditions that cause a trap. From the standpoint of hardware support for garbage collection, this microprocessor checks for the occurrence of write barriers, and notifies them using the `gc_notify` trap. This trap is triggered under certain conditions when to a reference field of some object (or an element of an array) is assigned a new reference. Our proposal includes the use and adaptation of existing hardware support. We use two types of write barriers: those typically used in generational collectors, based on the region to which the object belongs, and those typically used in incremental collectors, based on the colour of the object. A combination of both techniques, is further used.

The page-based barrier mechanism of picoJava-II was designed specifically to assist collectors based on the train algorithm, which traps when, within a memory area divided into a number of fixed-size spaces, which is also divided into a number of fixed-size *cars*, an object (X) references another object (Y) located in the same space but in a different car. We can use this mechanism to detect references across different MRs (i.e., inter-region references), by using only a memory space and considering that each car is a memory region. Configuring the `gc_notify` signal to trap when inter-region references, avoid to execute write barrier code for intra-region assignments.

The reference-based write barriers of picoJava-II was designed to implement incremental collectors based on the tri-colour algorithm like the one we use. Then, we must configure the `gc_notify` signal to trap when the application attempts to make a reference from a black object (X) to a white one (Y). This mechanism allows us to improve the performance of both the collector and the application by disabling write barriers execution to preserve the tri-colour invariant when disabling the collector. Similarly, we use the reference-based write barrier mechanism to detect when the root-set of the collector must be updated (i.e., when the X object is red and the Y object is not red). Also, in order to avoid illegal assignments of critical tasks to objects within the heap, we must configure the `gc_notify` signal to trap whether the Y object is red. In this way, reference-based write barriers avoid the execution of write barrier code when the object assignment does not attempt neither violate the tri-colour invariant, create a new root for the GC, or create a possible interaction of the GC and a critical task.

Since the conditions upon the `gc_notify` traps are different for critical and non-critical tasks, we must configure the reference-based write barrier each time that a task is scheduled to execute (i.e., at context-switch time).

3.2. The Write Barrier Trap Handler

When handling a write barrier, we can distinguish three main conditions depending on the MR of the referenced object (i.e., the Y object): *(A)* when it is within the heap, *(B)* when it is within the immortal region, and *(C)* when it is within a scoped region. Since the exception treatment and the configuration for the reference-based write barriers of the picoJava-II microprocessor is different depending on whether the task is critical or not, then, we analyze the above conditions for both critical tasks and non-critical tasks. Table III shows the actions that we must execute when an object Y is assigned to another object X.

- Condition *A*: (i.e., region(Y) = heap) is not allowed for critical tasks. For non-critical tasks (i.e., threads and high priority tasks), we must make the distinction depending on the colour of the object that makes the reference (i.e., the X object): Whether it is black and the Y object is white *(A.1)* we must avoid the tri-colour invariant violation by greying the Y object. And whether it is red *(A.2)*, we must maintain the root set of the collector by including the X object into it, and greying the Y object if it is white.
- Condition *B*: (i.e., region(Y) = immortal) is allowed and does not require any treatment.
- Condition *C*: (i.e., region(Y) = scoped), we make distinction depending on the MR of the X object: Whether it is within the immortal region or the heap *(C.1)*, we must avoid an illegal assignment by throwing the `IllegalAssignment()` exception. And whether it is within a scoped region different than the region to which the Y object belongs *(C.2)*, we must detect an illegal assignment by exploring the region stack associated to the active task.

Table III. Actions for inter-region references.

Category condition	X object region	Y object region	Action	
			Threads and Tasks	**Critical tasks**
A	heap	heap	tri-colour invariant	illegal assignment
	immortal	heap	update root-set	illegal assignment
	scoped	heap	update root set	illegal assignment
B	heap	immortal	allowed reference	allowed reference
	immortal	immortal	allowed reference	allowed reference
	scoped	immortal	allowed reference	allowed reference
C	heap	scoped	illegal assignment	illegal assignment
	immortal	scoped	illegal assignment	illegal assignment
	scoped	scoped	nested regions (When regions are different.)	nested regions (When regions are different.)

```
gc_notify_critical
  if (region(Y) = scoped)
      if (region(X) = scoped) nestedRegions(X,Y)     // C.2
      else illegalAssignment();                      // C.1
  if (region(Y) $\neq$ red) illegalAssignment();          // A
  priv_ret_from_trap                                 // B
```

Figure 9. Write barrier handler for critical tasks.

At this point there are two possibilities, the first one is to change the address of the exception vector value for the `gc_notify` exception at context-switch time, depending on the type of the scheduled task: critical (see Figure 9), and non-critical (see Figure 10). the other one consists to merge the handler for both critical and non-critical tasks. The former solution increases the context-switch overhead. However, it is more efficient than the latter, which introduce a write barrier overhead each time that a non-critical task makes an assignment to an object within the common heap. Our objective is to minimize the write barrier overhead, but also to prioritize critical tasks, therefore we chose the latter solution.

3.3. Modifying the Hardware Support

Note that two different mechanisms detect the aforementioned conditions: *(i)* the *B* and *C* conditions are detected by the page-based write barrier mechanism, *(ii)* the *A* and *A.1* conditions are detected by reference-based write barriers, and *(iii)* the *A.2* condition is detected by both mechanism reference-based and page-based write barriers. The hardware support of picoJava-II throws the `gc_notify` with priority level 14 for both reference-based and page-based write barriers. Since we must treat each condition in a different way, it is pretty interesting

```
gc_notify_non_critical
   if (region(Y) = scoped)
        if (region(X) = scoped) nestedRegions(X,Y)                              // C.2
        else illegalAssignment();                                               // C.1
   if (region(Y) $\neq$ red)       if (colour(X) = red) updateRootSet(X, Y)          // A.2
      else if ((colour(X) = black) and (colour(Y) = white)) greyObject(Y);      // A.1
   priv_ret_from_trap                                                          // B
```

Figure 10. Write barrier handler for non-critical tasks.

to make distinction by hardware whether the trap is caused by a reference-based condition or a paged-based condition, therefore we propose to introduce a new signal for distinguishing them The proposed modification further requires the addition the three following entries in the exception vectors table of the picoJava-II microprocessor: *(i)* the `gc_notify_0` which traps with the priority level 14 upon reference-based write barriers, *(ii)* the `gc_notify_1` which traps with the priority level 13 upon page-based write barriers, and *(iii)* the `gc_notify_1_0` which traps with the priority level 12 upon both conditions The introduction of these three exceptions improves the system performance because it avoids analyzing by software the cause of the `gc_notify` trap.

4. EVALUATING THE OVERHEAD

In this Section, we first review the three different write barrier implementations that we have proposed 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. A complementary analysis of the parameters characterizing the behavior of the GC can be found in [19], [29].

4.1. Write Barrier Implementations Solutions

4.1.0.1. Solution 1. Modifying the Java Interpreter. This solution consists in modifying the JVM by introducing the code given in Figure 11 in the interpretation of each bytecode whose function consists in assigning an object Y to another object X[‡].

4.1.0.2. 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 [18]. In this

---

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

```
if (region(Y) = scoped)
     if (region(X) = scoped) nestedRegions(X,Y)                                  // C.2
     else illegalAssignment();                                                   // C.1
if (region(Y) $\neq$ red)
   if ($\tau$ = critical) illegalAssignment()                                    // A
   else if (colour(X) = red) updateRootSet(X, Y)                                 // A.2
      else if ((colour(X) = black) and (colour(Y) = white)) greyObject(Y);       // A.1
priv_ret_from_trap                                                               // B
```

Figure 11. Write barrier handler for both critical and non-critical tasks.

```
gcnotify
   if (region(Y) = scoped)
        if (region(X) = scoped) nestedRegions(X,Y)                               // C.2
        else illegalAssignment();                                                // C.1
   if (region(Y) $\neq$ red)
      if ($\tau$ = critical) illegalAssignment();                                // A
      else if (colour(X) = red) updateRootSet(X, Y)                              // A.2
         else if ((colour(X) = black) and (colour(Y) = white)) greyObject(Y);    // A.1
   priv_ret_from_trap                                                            // B
```

Figure 12. Handling the `gc_notify` exception.

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. 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 12).

4.1.0.3. Solution 3. Modifying the Existing Hardware. This solution modifies the hardware support of picoJava-II to have three different traps (see Figure 13). In this solution, non-critical tasks cause the execution of: *(i)* the `gc_notify_1_0` 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 ( $\tau \neq$ critical) updateRootSet(X,Y) else illegalAssignment();    // A.2
    priv_ret_from_trap
gc_notify_1:
    if (region(Y) = scoped) nestedRegions(X,Y);                                 // C
    priv_ret_from_trap                                                          // B
gc_notify_0:
    if ( $\tau \neq$ critical) greyObject(Y) else illegalAssignment();          // A.1
    priv_ret_from_trap
```

Figure 13. Write barrier exception handlers.

Table IV. Memory reference behavior.

	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

4.2. Quantifying the Overhead.

All the objects created in Java are allocated within the JVM heap (i.e., dynamic memory, that in RTSJ may be either the heap or another MR); only primitive types are allocated in the runtime stack [27]. In most applications of the SPECjvm98 benchmark, suite[§], 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*. 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 [16]. Then, 15% (i.e., 0.45×0.35) of the bytecodes reference an object within the heap, where 30% make an assignment operation. Such as references to objects within MRs requires write barriers when assignment operations, 5% (i.e., 0.15×0.30) of the bytecodes require to execute a write barrier code.

To obtain the write barrier overhead for the solutions given in § 4.1, two measures are combined: (*i*) the number of events (*NEvents*), and (*ii*) the measured cost of the event (*ECost*). We also take into account the percentage of bytecodes requiring write barriers, which

---

[§] `http://www.spec.org/osg/jvm98`

has been evaluated as 5%. Then, we compute the total write barrier overhead introduced by both MRs and the GC:

$$MR_{Overhead} = NEvents_{MR} \times ECost_{MR} + NEvents_{scoped} \times ECost_{scoped}$$
$$GC_{Overhead} = NEvents_{GC} \times ECost_{GC} + NEvents_{incGC} \times ECost_{incGC}$$

Where $ECost_{MR}$, $ECost_{scoped}$, $ECost_{GC}$, and $ECost_{incGG}$ parameters can be obtained as:

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

4.3. Event parameters

To quantify the write barrier overhead, 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-colour invariant.

Notations:

We assume here that each object has an equal probability to being referenced.

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 the 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-colour 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-colour 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: $b \times (h - (b + g)) \leq b \times h - b^2$. 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., $h = \frac{h}{2} + w$).

Note that for critical tasks, the overhead due to the GC is 0 (i.e., $Event_{GC}$ and $Event_{incGC}$ are zero, otherwise the `IllegalAssignment()` exception raises). Then, we estimate the maximum probability to execute the write barrier code when a non-critical task makes an assignment, as given in Table V.

Table V. Max bound on write barrier events.

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

4.4. Cost parameters

We consider the event cost as the execution time expended by the introduced write barrier code 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. The write barrier cost is proportional to the number of evaluated conditions. The execution time taken by both the `greyObject(Y)` and the `updateRootSet(X, Y)` functions is considered as part of the GC overhead rather than as part of the write barrier overhead. For scoped regions, we must consider further the cost to have nested scoped levels, i.e., the cost to execute the `nestedRegions(X,Y)` function, that is proportional to the number of inner levels of the region to which the Y object belongs in the region-stack. Recall that in the first step of the algorithm, the region-stack is explored from the top to the bottom to find the region of the X object. Suppose that the number of explored levels is x and the region-stack have n levels. In the second step of the algorithm, the region-stack is explored from the region found in the previous step (i.e., the $n - x$ inner level) to found the region of the Y object. Suppose that the number of explored levels is y (i.e., it is found at the $n - x - y$ inner level). Since it is evident that $n >= 0$, then $x + y >= n$. We conclude taken n as the maximum bound of executed evaluations to check whether a region is outer than another one. Then, we bound the cost parameters as:

$$writeBarrierCost = maxConditions \times \frac{conditionCost}{assignmentCost}$$

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)`. The $conditionCost$ parameter is the execution time to evaluate a condition. Table VI gives the

Table VI. Evaluated conditions for write barrier.

maxConditions	**Solution 1 and 2**	**Solution 3**
$maxConditions_{MR}$	2	1
$maxConditions_{scoped}$	n	n
$maxConditions_{GC}$	3	1
$maxConditions_{incGC}$	2	1

maximum value for the number of evaluated conditions, where n is the maximum number of nested scoped levels.

Let $MR_{Overhead_i}$ and $GC_{Overhead_i}$ $(1 \leq i \leq 3)$ be the $MR_{Overhead}$ and $GC_{Overhead}$ parameters of each given solution, then:

$$MR_{Overhead_1} < (2 + n\frac{s}{m}) \times \frac{conditionCost}{assignmentCost}$$
$$MR_{Overhead_2} < (\frac{x}{m} + 0.25\frac{h}{m}) \times MR_{Overhead_1}$$
$$MR_{Overhead_3} < (\frac{i}{m} + (n+1)\frac{s}{m}) \times \frac{conditionCost}{assignmentCost}$$
$$GC_{Overhead_1} < (3 + 2(1 - \frac{x}{m})) \times \frac{conditionCost}{assignmentCost}.$$
$$GC_{Overhead_2} < (\frac{x}{m} + 0.25\frac{h}{m}) \times GC_{Overhead_1}$$
$$GC_{Overhead_3} < 1.2\frac{h}{m} \times \frac{conditionCost}{assignmentCost}$$

4.5. Comparison.

In Solution 1, the write barrier code is executed for all references. Solution 2 reduces the cost of write barriers for intra-region references to the cost to maintain the tri-colour 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-colour invariant. Solution 3 reduces also the cost for inter-region references, to the cost to detect both illegal assignments when the assigned object is outside the heap, and root-set updates when the referenced object is within the heap.

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.

5. IMPLEMENTATION DETAILS

We have modified the KVM garbage collector (Version 1.0.1) making it a stack-based tri-colour algorithm. We have implemented the `IncrementalGC` class within the KVM by modifying some files (i.e., 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). This class supports the method related with parameters characterizing the collector behavior (i.e., `getMinimumReclamationRate()`, `setReclamationRate()`, `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. RTSJ does not consider the write barrier overhead for MRs, then we add the `getWriteBarrierOverhead()` method to the `MemoryArea` 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` class, which identifies the write barrier cost to have `n` nested levels for scoped regions.

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 Ellis and Kovac benchmark that we can found in *http : //www.hpl.hp.com/personal/Hans_Boehm/gc.* 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*. 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:` 1, 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.

5.1. Memory Footprint

We have limited the number of regions to 256. Then, we need to add a word to the object header to include the following fields: `REGION_ID <11:4>`, `REGION_TYPE <3:2>`, and `COLOUR <1:0`. Where the `REGION_ID` field specifies the MR to which the object belongs, the `REGION_TYPE` specifies the MR type (e.g., 00 for the heap, 01 for immortal and 10 for scoped), and the COLOUR field specifies the colour of the object (e.g., 00 for black, 01 for grey, 10 for white, and 11 for red). This increases a word per object the memory consumption. Alternatively, we can modify the original header format of KVM objects (i.e., `SIZE <31:8>`, `TYPE <7:2>`, `MARK_BIT <1>`, and `STATIC_BIT <0>`) to support the colour and region of the object (i.e., `SIZE <31:17>`, `REGION_ID <16:10>`, `REGION_TYPE <9:8>`, `TYPE <7:2>`, and `COLOUR <1:0>`). The old `MARK_BIT` that is used by the original mark-and-sweep collector of the KVM to mark the object is not longer used because objects are market by colour, also the old `STATIC_BIT` is not used because it came from an old collector based on the copying algorithm that have been changed in order to make the KVM suitable to small devices. Note that the maximum size of the object has been reduced from

Table VII. **Garbage Collector Register** (`GC_CONFIG`).

Bits	**Field**	**Description**
31:21	`SPACE_MASK`	allows knowing if both the X and Y objects belong to the same space.
20:16	`CAR_MASK`	allows knowing if both the X and Y objects belong to the same *car*.
15:0	`WB_VECTOR` *(Write Barrier Vector)*	if the corresponding bit is set, the bytecodes **`putstatic`**, **`aputfield_quick`**, **`aputstatic_quick`**, **`aastore`**, and **`aastore_quick`** signal a **`gc_notify`** trap.

16 Mbytes to 32 Kbytes; given the small average object size that the specJVM applications present (i.e., about 32 Bytes), we optimize for small objects. We also maintain a region-structure of 4 words for each MR object in the system with the following format: `REGION_ID <63:56>`, `OUTER_REGION <55:48>`, `REFERENCE_COUNTER <47:42>`, `REGION_TYPE <41:40>`, `INITIAL_SIZE <39:25>`, and `MAXIMUN_SIZE <24:0>`, which increases the memory footprint as maximum of 2 Kbytes. Note that these region-structures forms a scope-three where the heap is the root and immortal regions are not included.

5.2. Configuration

In picoJava-II, the conditions under which the write barrier trap is generated are governed by the values of the *Garbage Collection Register* (`GC_CONFIG`), which configuration is summarized in Table VII. The object reference in picoJava-II has 4 fields (i.e., the `GC_TAG` field determines whether to signal a write barrier GC trap, the `ADDRESS` field contains the address of the header object, the `X` field indicates whether the object is an array, and the `H` indicates if the object is referenced directly or indirectly). The `GC_TAG`, `X`, and `H` fields are masked before the reference is used as an address, and also in comparison instructions (i.e., **`if_acmpeq`** and **`if_acmpne`**). For direct references, the object instance variables start one word after the header. For indirect references, we must go through the handle to access the object. Then, the $< 31 - 30 >$ and the $< 1 - 0 >$ bits of the address are not used to access memory; the 3 *GBytes* with greater addresses are reserved for the system and cannot be used by the application, and object headers are mapped into addresses that are multiples of 4.

Since, we map each RTSJ memory region object to a car of picoJava-II, and the `CAR` field of the object reference indicates the MR object within which the object is allocated. Then, the maximum number of MRs is 32, and each region is composed of several pages, where the maximum number of pages is 2048. The page size is between 16 *KBytes* and 512 *KBytes*, depending on the number of MRs (i.e., if we have 32 regions, the page size is 16 *KBytes*, whereas if we have only 1 region the page size is 512 *KBytes*). For each MR, we maintain a list of pages or a map of bits indicating which pages are assigned to the region.

To have a single space, we configure the `SPACE_MASK` field of the `GC_CONFIG` register with the 0000.0000.000 value, which masks the $< 29 : 19 >$ bits in the `ADDRESS` field of the object reference. Considering the `CAR_MASK` field of the object reference configured as 1.1111, the same car is repeated each 512*KBytes* in the memory map, which means every 32 pages, each one

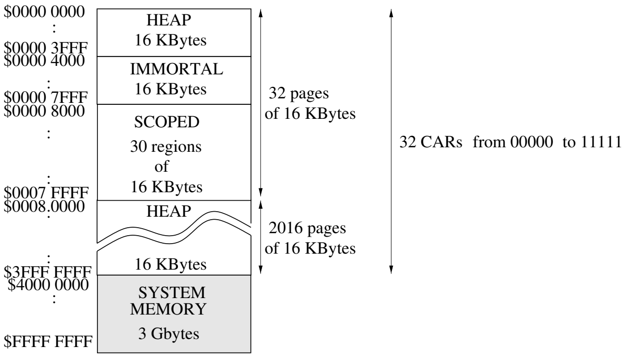

Figure 14. Memory map for MRs (`CAR_MASK`=11111).

belonging to a different MR (see Figure 14). More specifically, `page_0` (address $\$000000000$-$\$00003FFF$) belongs to the heap, `page_1` (address $\$00004000$-$\$00007FFF$) belongs to the immortal memory, and pages from `page_2` to `page_31` belong to 30 different scoped MRs. This mapping is repeated along the overall address space, i.e., `page_32` belongs to the heap, `page_33` belongs to the immortal memory, and pages from `page_34` to `page_63` belong to 30 different (immortal physical or scoped) MRs.

Consider the following colour codes: 11, 10, 01, and 00 respectively denote black, grey, red, and white objects. We obtain the following configuration values:

- Reference-based write barriers for critical tasks avoid accesses to objects within the heap, detecting assignments to objects within the heap (i.e., to black, grey, or white objects) which give the following combinations: 1111 (black to black), 1110, 1100, 1011, 1010, 1000
  0111, 0110, 0100 0011, 0010, 0000 (i.e., bits 15, 14, 12, 11, 10, 8, 7, 6, 4, 3, 2, and 0). Then, we must configure the `WR_VECTOR` with the 1101.1101.0000.1101 value (i.e., $\$BB0B$).
- Reference-based write barriers for non-critical tasks enable to preserve the tri-colour invariant (i.e., the 1100 combination) and to maintain the root-set (i.e., 0111, 0110, 0100 combinations). Then, we must configure the `WR_VECTOR` field with the value $\$10C0$ (i.e., 0001.0000.1101.0000).

In order to use the write barrier hardware aid of picoJava-II, as we have described, we introduce: two routines to configure write barriers for both critical and non-critical tasks (see Figure VIII). Where the `priv_read_gc_config`, and `priv_write_gc_config` extended bytecodes allowing access to the `GC_CONFIG` register.

Table VIII. Configuring write barriers.

```
configureWriteBarrierCriticalTasks:
        spush 0xBBBB  // reference-based
        seti 0x001F   // page-based
        priv_write_gc_config
        priv_ret_from_trap
```

```
configureWriteBarrierNonCriticalTasks:
        spush 0x10C0  // reference-based
        seti 0x001F   // page-based
        priv_write_gc_config
        priv_ret_from_trap
```

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. This specification imposes restricted assignments rules that keep longer-lived objects from referencing objects in scoped memory, which possibly have a shorter life. This requires run-time checks for each assignment, which introduces a high overhead. In this paper, we have proposed a solution to the realization of the abstract memory model introduced by the RTSJ specification. In particular, garbage collection in the heap complies with real-time constraints by using write barriers to maintain both the root-set and the tri-colour invariant.

In our solution, the detection of illegal assignments related with memory regions and illegal accesses related with critical tasks, is made dynamically by introducing a write barrier mechanism based on a region-stack associated to the active task. We improve the performance of our solution by using the write barrier support of the picoJava-II microprocessor, as proposed in [18]. In this solution, an exception trap for inter-region references and must be configured at context-switch time depending on the scheduled task, because non-critical tasks trap when a white object is assigned to a black one, whereas critical tasks traps when the assigned object is not red. We also propose to modify the hardware support of picoJava-II to have three different traps: *(i)* to preserve the root-set of the collector *(ii)* to detect illegal assignments and *(iii)* to preserve the tri-colour invariant.

Hardware-based implementations are efficient, but quite inflexible. We must configure the system to determine the virtual region memory map. In addition, our solution requires the size of a region to be a multiple of the car size, which may possibly introduce internal fragmentation. Finally, for a `VTMemory` scoped region that can change its size up to its *maximumSize*, the additional memory must be assigned in terms of cars. This problem can be unpractical for classes dealing with I/O mapped memory (e.g., `ScopedPhysicalMemory`), which specify in their constructor not only the *size* of the region, but also the *base* address.

Another problem with is that we omit write barriers in native code, which may be addressed using either of the two following solutions: forcing the native code to register their writes explicitly, or using virtual memory protection to detect and register changes. The latter solution needs further investigation because it is not trivial to combine real-time bounded collection with barriers supported in the MMU.

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. In this way, the application runs with static memory. The prototype given in [15] avoids both collectors, those collecting objects within the heap and those collecting unused scoped regions, which facilitates an accurate pre-runtime analysis of the memory management behavior. In general, static analysis is used to determine worst-case, and performance measures are used to determine average-case. It is also advisory to use a combination of both, analytical and empirical techniques, to evaluate resource usage and to optimize resource consumption.

ACKNOWLEDGEMENTS: This work has been partially founded founded by Texas Instruments. The authors gratefully acknwledge the comments and guidance of Miguel Angel de Miguel.

REFERENCES

1. N. Fresko and W. Foote. Proposed Java Real Time Extension Specification. http://www.sdct.itl.nist.gov/ carnahan/real-time/sun/index.html, 1998.
2. J Consortium Inc. Core Real-Time Extensions for the Java Platform. http://www.j-consortium.org/rtjwg.html, August, 1999.
3. The Real-Time for Java Expert Group (RTJEG). Real-Time Specification for Java, http://www.rtj.org, June, 2000.
4. M.T. Higuera-Toledano, V. Issarny, M. Banâtre, 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)*, IEEE, March, 2000.
5. Sun Microsystems. KVM Technical Specification. http://java.sun.com, May, 2000.
6. H. McGhan and M. O'Connor. picoJava: a Direct Execution Engine for Java Bytecode. *IEEE Computer*, **vol. 31, number 2, October, 1998, pag. 22-30**.
7. G. Bollella and J. Gosling. The Real-Time Specification for Java. *IEEE Computer*, **June, 2000**.
8. The Real-Time for Java Expert Group (RTJEG). Real-Time Specification for Java. http://www.rtj.org, June, 2000.
9. Standard Performance Evaluation Council. SPEC JVM98 benchmarks. http://www.spec.org/osg/jvm98, 1998.
10. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholtenand, and E.F.M. Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, **vol. 21, number 11, November 1978**.
11. P.R. Wilson and M.S. Johnstone. Real-Time Non-Copying Garbage Collection. In *Workshop on Garbage Collection and Memory Management.* ACM OOPSLA", September, 1993.
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*, December, 1998.
13. D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proc. of the Conference of Programming Language Design and Implementation (PLDI).* ACM SIGPLAN, June, 1998.
14. M.T. Higuera, V. Issarny, M. Banâtre, G. Cabillic, J.P. Lesot, and F. Parain. Memory Management for Real-time Java: an Efficient Solution using Hardware Support. *Real-Time Systems journal,Kluber Academic Publishers.* http://www.kluweronline.com/issn/0922-6443/contents, **to appear**.
15. 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.
16. J.S. Kim and Y. Hsu. Memory System Behavior of Java Programs: Methodology and Analysis. In *Proc. of the ACM Java Grande 2000 Conference.* June, 2000.
17. P.R. Wilson. Uniprocessor Garbage Collection Techniques. ftp://ftp.cs.utexas.edu/pub/garbage/bigsur.ps.

18. M.T. Higuera-Toledano, V. Issarny, M. Banâtre, G. Cabillic, J.P. Lesot, and F. Parain. Region-based Memory Management for Real-time Java. In *Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, May, 2000.
19. M.T. Higuera-Toledano 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, May, 2000.
20. M.T. Higuera and M.A. de Miguel. Dynamic Detection of Access Errors and Illegal References in RTSJ. In *Proc. Of the 8th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*. IEEE September 2002.
21. M. Tofte. Implementing the Call-by-Value Lambda-Calculus using a Stack of Regions. In *Proc. of the Conference of programming Languages Design and Implementation (PLDI)*. ACM SIGPLAN, January 1994.
22. D.J. Cannarozzi, M.P. Plezbert, and R.K. Cytron. Contamined Garbage Collection. In *Proc. of the Conference of programming Languages Design and Implementation (PLDI*. ACM SIGPLAN, May 2000.
23. K. Nilsen. Adding Real-Time Capabilities to Java. *Communications of the ACM*, **vol. 41, number 6, June 1998**.
24. C.E. McDowell Reducing Garbage in Java. http://www.cs.Berkeley.edu, 1997.
25. 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.
26. B. Zorn. Barrier Methods for Garbage Collection. http://www.cs.colourado, November, 1990.
27. D. Gay and B. Steensgaard. Stack Allocating Objects in Java. http://www.research.microsoft.com/apl, 1998
28. Sun Microsystems. picoJava-II Programmer's Reference Manual. http://www.sun.com/microelectronics/picoJava, March, 1999.
29. M.T. Higuera-Toledano. Memory Management Solutions for Real-time Java. INRIA-Rocquencourt, Mars, 2002.