Real-Time Systems, 26, 63–87, 2004

Memory Management for Real-Time Java: An Efficient Solution using Hardware Support*

TERESA HIGUERA teresa.higuera@inria.fr
VALÉRIE ISSARNY
INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153, France

MICHEL BANÂTRE
JEAN-PHILIPPE LESOT
FRÉDÉRIC PARAIN
GILBERT CABILLIC
INRIA-IRISA, Campus de Beaulieu, 35032 Rennes Cédex, France

Abstract. This paper addresses the issue of improving the performance of memory management for real-time Java applications, building upon the real-time specification for Java (RTSJ) from the Real-Time Java Expert Group. In a first step, a collecting dynamic memory solution including both a real-time garbage collector and region-based memory management, is proposed. A thorough analysis of the parameters influencing the performance of write barriers in memory management, together with ways of improvement are then presented. Finally, the implementation of a memory management solution compliant with the RTSJ and integrating the proposed improvements is sketched.

Keywords: Java, real-time, embedded, garbage collection, memory regions, write barriers, performance

1. Introduction

The Java environment provides attributes that make it a powerful platform to develop embedded real-time applications. However, it presents some important lacks regarding its use in this kind of systems (Higuera et al., 2000). The original Java platform was designed for computers with RAM memory (personal computers or workstations). Then, other Java platforms were defined to support real-time and embedded systems. In particular JavaCard (Sun, 1998), EmbeddedJava (Sun, 1999) and PersonalJava (Sun, 1998) are three different platforms from Sun Microsystems. But none of these Java adaptations integrate the techniques and methods that real-time systems require.

The National Institute of Standards and Technology (NIST), has produced a basic requirements document (Carnahan, 1998) for a standard real-time Java API extension. Solutions that comply with this document are the real-time specification for Java (RTSJ) (RTJEG, 2000) and the real-time core extension for the Java platform (RT Core) (J Consortium, 1999). There are some other solutions that were introduced before the NIST document. The simplest one is a prototype introducing tasks support over RT-Mach (RT-Threads) (Miyoshi et al., 1997). Another proposal is the portable executive for reliable

* This work has been partially funded by Texas Instruments.

Table 1. Comparison of studied solutions.

	(i)	(ii)	(iii)	(iv)	(v)	(vi)
RTSJ	A	A	A	A	M	A
RT Core	A	A	A	M	M	A
PERC	A	A	M	M	A	A
RT-Threads	A	A	—	—	M	M
CTJ	A	—	—	—	—	—
GVM	M	A	—	—	M	—
Aj-100	A	A	A	M	M	M

Notes. We use A, M, and—to respectively mean that the corresponding issue is addressed in detail, only partly addressed, and not addressed.

control (PERC) (Nilsen, 1998), that is close to the J Consortium solution, the latter being actually an evolution of PERC. A very different solution is the communication threads for Java (CTJ) (Hilderink, 1998), that is based on the CSP algebra, the Occam2 language and the Transputer microprocessor. Other solutions integrate the JVM into the operating system such as the GVM (Bak et al., 1998), a prototype centered around resource management. Another option to improve the performance of Java is to integrate the JVM in a microprocessor as the Aj-100 (Hardin, 2001), which implements the entire JVM instruction set in silicon and directly supports the Java thread model in hardware.

We have analyzed and studied how the above solutions resolve the problems that Java presents to effectively support embedded real-time applications (Higuera et al., 2000). We have divided these problems in the following categories: (i) the inability to access the underlying hardware, (ii) the unspecified behavior for thread scheduling, (iii) synchronization that requires stronger semantics, (iv) the inability to handle events, (v) the inability to specify resources, and (vi) dynamic memory management. A comparison of the aforementioned solutions is summarized in Table 1.

From our point of view, the RTSJ constitutes the most adequate solution for real-time systems in general, and a hardware support such as Aj-100 enables improving the system's performance. In the context of the activities of the Solidor group at INRIA, we are currently developing a Java-based software environment accounting for embedded real-time (Issarny et al., 2000). This paper focuses on how to make Java memory management real-time while accounting for relevant Java specifications: the RTSJ, the KVM (Sun. 1999) targeting limited-resource and network connected devices, and the picoJava microprocessor core (Sun, 1999).

1.1. Background

Implicit garbage collection has always been recognized as a beneficial support from the standpoint of promoting the development of robust programs. However, this comes along with overhead regarding both execution time and memory consumption, which makes (implicit) garbage collection poorly suited for small-sized embedded real-time systems. This must not lead to undertake the unsafe primitive solution that consists in letting the

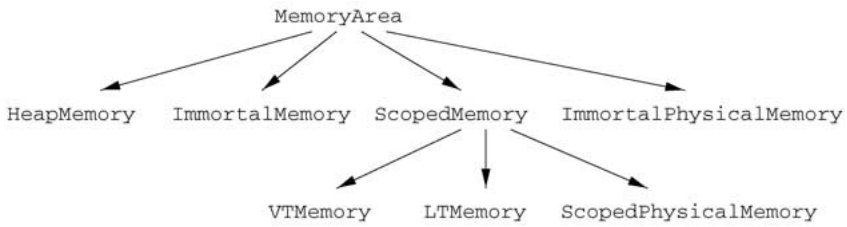


Figure 1. The `MemoryArea` hierarchy: Whereas `ImmortalMemory`, `HeapMemory`, and `ImmortalPhysicalMemory` objects end with the application, the life of `ScopedMemory` objects depend on the program control flow.

application programmer to explicit deal with memory reclamation. As an alternative, region-based memory allocation enables grouping related objects within a region. This is an intermediate solution between explicit memory deallocation (e.g., `free`() in C) and garbage collection. Application of the two above implicit strategies has been studied in the context of Java, which are combined in RTSJ. The `MemoryArea` abstract class supports the region paradigm in RTSJ through the following three kinds of regions (see Figure 1): (i) immortal memory contains objects whose lifetime ends only when the JVM terminates (i.e., supported by the `ImmortalMemory`[1] and the `ImmortalPhysicalMemory` classes); (ii) (nested) scoped memory 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` and the `ScopedPhysicalMemory`[2] classes) on the time taken to create objects; and (iii) the conventional heap (i.e., supported by the `HeapMemory` class).

Whereas there are only one `ImmortalMemory` and one `HeapMemory` objects in the system, several `ImmortalPhysicalMemory` and `ScopedMemory` objects can exist. Then, an application can allocate objects into the heap, the immortal region, or several scoped regions. Several related threads, possibly real-time, can share a memory region, and the region must be active until the last thread has exited. The default region is the Java heap. Allocations outside the active region can be performed by the `newInstance`() methods or the `newArray`() as shown in Figure 2.

1.2. Paper Organization

In the context of RTSJ, this paper proposes such a study, focusing on minimizing the execution time overhead caused by write barriers in implicit memory reclamation. We first present basic modifications to the garbage collector of a Java VM in order to make it compatible with real-time tasks execution, and a possible implementation of the memory region abstraction presented by the RTSJ (Section 2). A thorough analysis of the parameters influencing the performance of write barriers is presented regarding the management of memory regions (Section 3). A solution improving the write barrier performance of both memory regions and the collector is then given (Section 4). Results of this analysis are further exploited to derive a memory management solution for a Java

```
import javax.realtime;

class Allocator implements Runnable {
     public void run() {
        HeapMemory.instance().newArray(Integer, 10);
        int[] x = new int[20];
        ..................... // do some stuff
        }
}

class RegionUseExample {
     public static void main (String[] args) {
        ScopedMemory myRegion = new VTMemory(1024, 2*1024);
        RealtimeThread task = new RealtimeThread(
                                  null, null,
                                  new MemoryParameters(1024, 0),
                                  myRegion, null, new Allocator());
        task.start();
        }
}
```

Figure 2. Using memory regions in RTSJ: This code shows a real-time thread, which allocates an array of 10 integers in the heap, and another of 20 integers in the MR called `myRegion`.

environment aimed at wireless PDAs, which we are experimenting through adaptation of the KVM (Section 5). Finally, a summary of our contribution together with an overview of our ongoing and future research work towards offering an overall memory management solution for next-generation wireless PDAs conclude this paper (Section 6).

2. The Basic Approach

From a real-time perspective, the garbage collector (GC) introduces unpredictable pauses that are not tolerated by real-time task. Real-time collectors eliminate this problem but introduce a high overhead. An intermediate approach is to use memory regions (MRs) within which allocation and deallocation are customized, and also space locality is improved. Note that both collection strategies are complementary: a GC may be used within some regions in order to limit their size, while the use of regions allows reducing the runtime overhead due to GC.

In this section, we propose a possible implementation of memory reclamation for Java that is compliant with the RTSJ specification (Bollella and Gosling, 2000). The proposed solution addresses real-time constraints of the GC in the heap, and the problem associated with inter-region references introduced by MRs (i.e., external references, illegal accesses, and illegal references).

2.1. GC Strategy

There are some important considerations when choosing a real-time GC strategy. Among them are space costs, barrier costs, and available compiler support. Copying GCs require doubling the memory space, because all the objects must be copied during GC execution. Non-copying GCs do not require this extra space, but are subject to fragmentation. We specifically base our solution on the incremental non-copying collector called treadmill (Baker, 1991). Since this allows the application to execute while the GC has been launched, a mechanism, called read barrier, is used to keep the state of the GC consistent by coordinating the execution of the GC and of the application. The basic algorithm is as follows (see Figure 3): an object is colored white when not reached by the GC, black when reached, and gray when it has been reached, but its descendants may not be (i.e., they are white). Gray objects make a wavefront, separating the white (unreached) from the black (reached) objects, and the application must preserve the tri-color invariant that no black objects have a pointer to a white object, which is achieved using read barriers (i.e., the white object is grayed when it is accessed). The collection is completed when there are no more gray objects. All the white objects can then be recycled and all the black objects become white after the recycling phase. In this process, objects that must execute the `finalize()` method[3] are moved to a finalize-list as in the Kaffe JVM (Petit-Bianco and Tromey, 1998).

To coordinate the application and the GC, we use write barriers (i.e., a white object is greyed when the application creates a pointer from a black object) instead of read barriers (Wilcon and Johnstone, 1993). This decision is motivated by the fact that write barriers are more efficient than read barriers,[4] and that the resulting collector can further be easily extended to generational, distributed, and parallel collection. Then, violation of the tri-color invariant must be checked when executing instructions that store references within

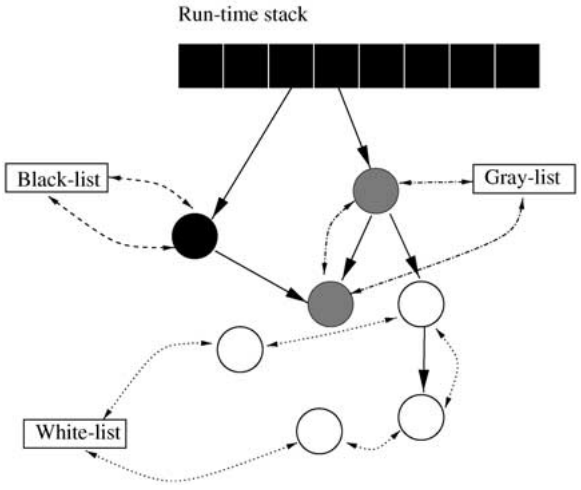


Figure 3. The GC strategy: to maintain list coherency, all the operations on shared lists must be *synchronized* with the application threads.

other objects (or arrays) (i.e., when executing `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, or `aastore_quick` bytecodes).

2.2. Dealing with Fragmentation

In general, memory fragmentation is not a severe problem, but for embedded systems (like PDAs) the amount of memory is normally small. Additionally, if the application only allocates objects with small size, acceptable worst-case bounds can be given. For objects with large size, strategies such as fragmenting the object into smaller-size chunks may be used. Another strategy relies on occasionally running a compacting GC, which implies some degradation of real-time guarantees. Since in Java, the size of reachable objects does not change much and is rather small (see Table 2 for applications from SPECjvm98; SPEC, 1998), the average cost of each compactation phase is constant.

Normally, to reduce the cost of object relocation, each object has a non-moving handle. Compaction is made in two phases: the object space is first compacted, and the handles are then updated. In this way, relocating objects is transparent to the application program, which always accesses objects using their non-moving handle. Given the small average object size (i.e., 36 Bytes), it appears imperative to keep the number of header words to a minimum. For example, a header of only a word will consume about 11% of the total memory heap, whereas if the header has two words, this consumption will increase to 20%. Hence, for some applications, it can be interesting to not use a compacting phase, avoiding handles.

Since objects outside the heap are not moved, we can improve the performance by avoiding the handle of these objects, and hence use direct reference objects. Eliminating handles improves also the memory consumption in a word per object. Note that this strategy is always possible even if the GC has a compactation phase requiring handles for objects within the heap. Suppressing handles in the heap improves the performance of both the application and the collector.

Table 2. Object characterization: number of objects allocated by each application, average object size in Bytes, average object lifetime in millions of references, and average number of references per each object.

	Allocated objects	Size	Lifetime	References
JESS	8,131,609	40	1.3	87
DB	3,262,899	31	129.5	449
JAVAC	6,244,896	36	55.2	116
MTRT	6,695,116	25	10.8	86
JACK	6,955,528	31	1.6	147

2.3. External References

Since objects allocated within immortal or scoped regions may contain references to objects in the heap, the collector must take into account these external references, adding them to its reachability graph. To facilitate this task, we color black each object allocated outside the heap (see Figure 4). In this way, a reference from an object allocated in a region (i.e., black) to an object in the heap that is still not reached (i.e., white) is treated as a write barrier (i.e., the white object is grayed to be reached by the GC). Black objects outside the heap are considered as root by the GC. This requires that all objects of all the MRs must be explored by the collector, which introduces high overhead.

Scoped regions can be nested. A safe region implementation requires that a region gets deleted only if there is no external reference to it. This problem has been solved in RTSJ by using a reference-counter for each region which keeps track of the use of the region by threads, and a reference-counting GC collects scoped memory regions when their reference-counter reaches 0. The reference-counter is increased upon entering a new scope through the `enter`() method, the creation of a real-time thread with a scoped region, and the opening of an inner scope. It is decreased when an inner scope returns from its `enter`() method or when the real-time thread using the scoped region exits.

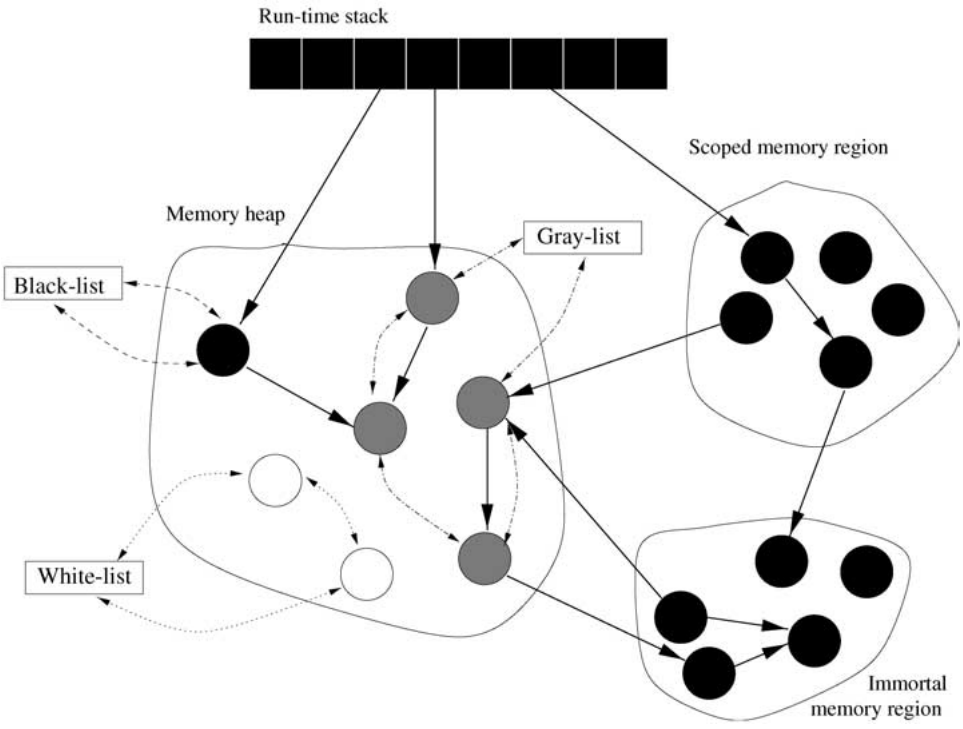


Figure 4. External references: objects within the heap referenced from immortal or scoped regions must be considered reachable by the GC.

2.4. Illegal Accesses

RTSJ makes distinction between three main kinds of tasks: (i) critical tasks that cannot tolerate preemption latencies, (ii) high-priority tasks that cannot tolerate unbounded preemption latencies, and (iii) low-priority tasks that are tolerant with the GC.[5] Whereas high-priority tasks require a real-time GC (i.e., the behavior of the algorithm given in Section 2.1 must be deterministic), critical tasks must not be affected by the GC, and as consequence cannot access any object within the heap. To detect these illegal accesses, we introduce a fourth color (e.g., red) meaning that the object cannot access objects within the heap (Figure 5).

An access from a red object allocated for a critical task to another object allocated in the heap (i.e., white, black, or gray) causes a `MemoryAcessError` exception. Illegal accesses from a red object to an object within the heap must be checked when executing instructions that load a reference to an object or array (i.e., at every read barrier, those bytecodes causing a load operation: `getfield`, `getstatic`, `agetfield_quick`, `agetstatic_quick`, or `aaload` and those causing a load-store operation: `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, or `aastore_quick`).

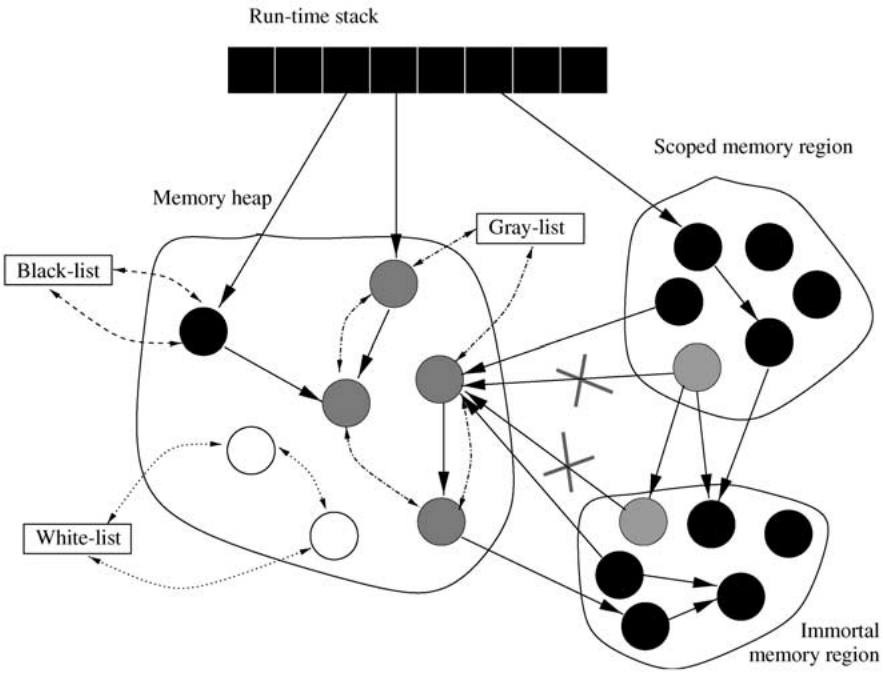


Figure 5. Illegal accesses: red objects cannot access objects within the heap.

2.5. Illegal Asignments

The lifetime of objects allocated in scoped regions is governed by the control flow (see Figure 6). To maintain the safety of Java and avoid dangling references, objects in a scoped MR can only reference objects within an outer region, within the heap, or within immortal memory. And, objects within the heap or within the immortal region cannot reference objects within a scoped region. In our example, an object within the scoped region associate which task `t1` can only be referenced by objects allocated within the same region or within an inner region (see Figure 7), i.e., within the scoped region associated which task `t3`.

To detect illegal references, we propose a stack-based algorithm, which associates a region stack with each thread. Note that references from objects within the heap, an

```
class ScopedRegionExample {

    static Runnable codeY = new Runnable() {
          public void run() {
             Y y1 = new Y();
             Y y2 = new Y();
             LTMemory r3 = new LTMemory(32768, 32768);
             RealtimeThread t3 = new NoHeapRealtimeThread(r3, null, codeX);
             t3.start();
             // do some stuff
          }
     };

     static Runnable codeX = new Runnable() {
       public void run() {
          X x1 = new X();
          X x2 = new X();
          // do some stuff
       }
    };

     public static void main(String args[]) {
         VTMemory r1 = new VTMemory(32768, 65536);
         VTMemory r2 = new VTMemory(32768, 65536);
         RealtimeThread t1 = new RealtimeThread(r1, null, codeY);
         RealtimeThread t2 = new RealtimeThread(r2, null, codeX);
         t1.start();
         t2.start();
     }
}
```

Figure 6. Using scoped memory regions in RTSJ. This code shows three tasks (`t1`, `t2`, and `t3`), whose one is critical (`t3`), which allocate allocating objects in a MR (i.e., `t1` within `r1`, `t2` within `r2`, and `t3` within `r3`).

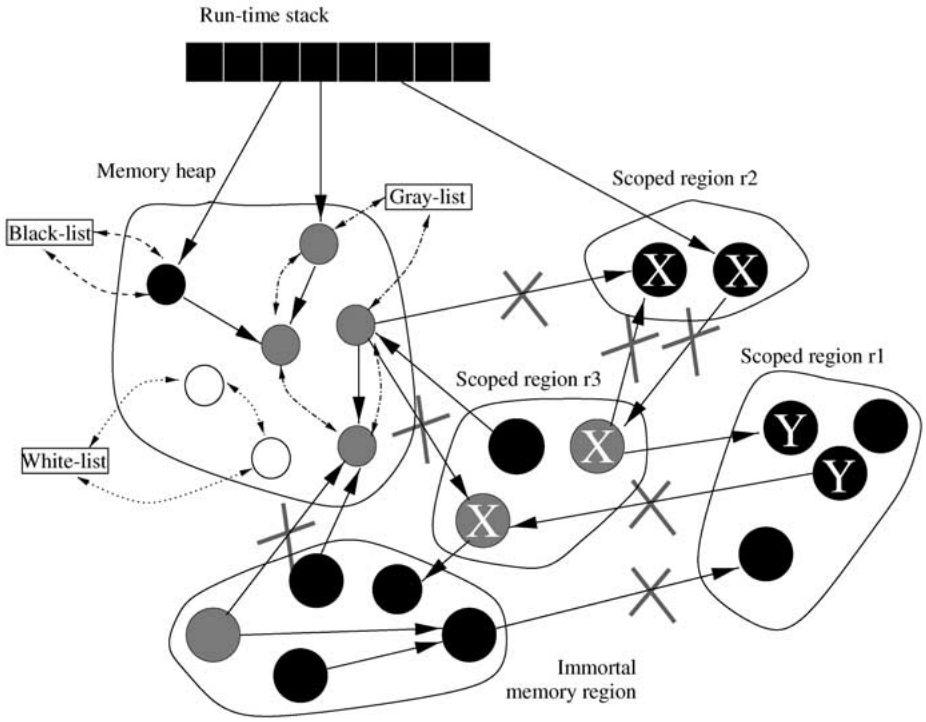


Figure 7. Illegal references: an object within a scoped region can only be referenced by objects from the same region or within an inner region (the region `r3` is inner to the region `r1` from the code of Figure 6.

immortal MR, or a scoped MR, to objects within the heap or an immortal memory are allways allowed. Then the region stack of a thread supports scoped regions where the thread can reference objects. Figure 8 shows the region stack associated with the tasks of our example. This mechanism was introduced in Higuera et al., (2001) and is similar to the contaminated GC, a Java stack-based collector given in Cannarozzi et al. (2000), which collects objects when the scoped associated to a control flow ends.

We detail below how to support such a functionality:

- When an object is created, it is associated with the scope of the active region.

- When removing a region, the top of the region stack is adjusted, and it is sure that there is no object dependent on an older scoped region. Note that the immortal memory and the heap, which are at the bottom of the stack end with the application, and as consequence are never removed from the stack.

Illegal inter-region references causes an `IllegalAsignmentError` exception and must be checked when executing instructions that store references to objects outside the heap within other objects (or arrays) (i.e., when executing `putfield`, `aputfield_quick`, `aastore`, or `aastore_quick` bytecodes). The `putfield` (`aputfield_quick`) instruction

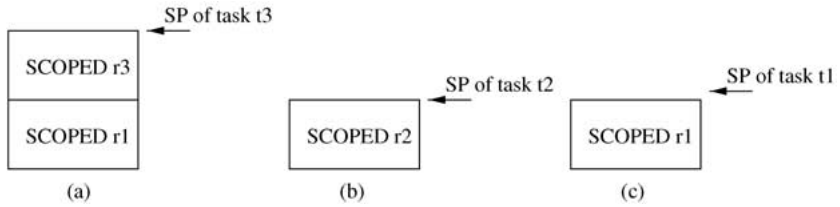


Figure 8. Region stack: Each task has an associated region stack: (a) region stack of the critical task `t1`, (b) region stack of task `t2`, and (c) region stack of task `t3`.

causes the object X to reference Y, whereas the `aastore` (`aastore_quick`) instruction stores a reference Y into an array X of references. If the object Y is within a scoped region, the region of the object X must be inner to the region of Y. This check can be made by using the region stack, from the region of X down to the region of Y (an outer region). If the region of Y is not found in the stack (i.e., the heap that is the outest region and hence the bottom of the stack is reached), this is notified by throwing an exception. Whereas the `putstatic`(`aputstatic_quick`) instruction causes a reference to an object whithin the outermost region (i.e., the `HeapMemory object`). Then, checking the stack is not needed.

3. Analyzing the Performance of Region Management

In general, the management of memory regions introduces overhead, which we characterize in this section. The cost to maintain MRs is considered as a fraction of the total program execution time. To estimate the time overhead of different implementations, two measures are combined: (i) the number of events that occur during the execution of a program, and (ii) the measured cost of the event. Then, the region overhead is given by dividing the application execution time with the number of events and the cost per event. The region implementation given in Gay and Aiken (1998) presents a time overhead that is constant per instruction executed. The RTSJ imposes strict rules on assignments to or from regions. The JVM must detect illegal accesses and assignments and throw an exception when they occur. Also, the collector must scan regions for references to objects within the heap which introduces an additional overhead.

3.1. Memory Management Overhead

Each MR is managed so as to embed objects that are related regarding associated lifetime and real-time requirements. GC within the heap relies on the (real-time) collector of the JVM. The immortal memory region is never subject to GC and may be exploited by critical tasks. And scoped region get collected as a whole once it is no longer used and may or may not be subject to internal real-time GC depending on their temporal properties.[6] Then, the overall cost introduced by region management is given by the cost associated with: region allocation, reference counter updates, and region deletion. The

time cost to allocate a new region is always constant. Note that by collecting regions, problems associated with reference counting collectors are solved: the space to store reference counters is minimal, and there cannot be cycles among regions. Before cleaning a region, the `finalize`() method of all the objects in the region must be executed, and it cannot be reused until all the finalizes execute to completion. Problems with the `finalize`() method are resolved by adding to the finalize-list of the GC, all the objects within the terminated region that are pending to execute this method.

From a real-time perspective, regions give predictable performance, since the cost of every allocation operation is easily bounded. Assuming that objects in a `ScopedMemory` region are not subject to GC and may not be moved,[7] 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. Whereas an allocation in a `VTMemory` region may take variable time, the time taken in a `LTMemory` region is linear to the object size. Then, the memory space for a `LTMemory` region must be continuous,[8] and its size is further specified upon creation, remaining fixed over its lifetime. Since objects allocated in `LTMemory` regions are not garbage collected the allocation time is proportional to the object size,[9] it is safe to associate this subclass with critical tasks. However, an instance of `VTMemory` is created with an initial size and may grow up to a given maximum size.

3.2. Region Barrier Overhead

Experimental measures indicate that in Lisp programs the references to the heap (as opposed to the runtime stack) account for an average of 12% of all executed instructions (Zorn, 1990). However, all the objects created in Java are allocated in the heap, only primitive types are allocated in the runtime stack (Gay and Steensgaard, 1998). In most applications of the SPECjvm98 benchmark suite,[10] less than half of the references are to the heap memory (i.e., 45%), the other half is to either the Java or the C stack (see Table 3), and about 35% of the total executed instructions are memory references (Kim and Hsu, 2000), where typically 70% are load operations and 30% store operations.

Then, 15% (i.e., 0.45 $*$ 0.35) of instructions executed by a Java application is a reference into the heap or another memory region, where 11% require read barriers executing the code of Figure 9 for checking illegal accesses (see Section 2.4). Thus we estimate the read barrier overhead as 0.11 $*$ read-Barrier, where the readBarrier

Table 3. Memory reference characteristics: number of instructions executed for each application, heap memory allocated in KB, and average allocation rate in Bytes/(1000 instructions).

	Executed instructions	Data references	Percentage of references into the heap
JESS	$9{,}168 \times 10^6$	$1{,}798 \times 10^6$	39.40
DB	712×10^6	$3{,}211 \times 10^6$	45.61
JAVAC	$7{,}717 \times 10^6$	$2{,}515 \times 10^6$	28.70
MTRT	$3{,}917 \times 10^6$	$1{,}129 \times 10^6$	50.97
JACK	$6{,}553 \times 10^6$	$2{,}014 \times 10^6$	50.74

```
readBarrierMR:
   if ((region(Y) = heap) and (color(X) = red)) goto memoryAccesError;
end_readBarrier:
```

Figure 9. Read barrier code for MRs.

```
writeBarrierMR:
   if (region(Y) = scoped)
         if (not checkNestedRegions(X, Y)) goto illegalAssignmentError;
   if ((region(Y) = heap) and (color(X) = red)) goto memoryAccesError;
end_writeBarrier:
```

Figure 10. Write barrier code for MRs: the `checkNestedRegions()` function is an implementation of the algorithm given in Section 2.5.

parameter is the percentage of the number of instructions executed by the code taged as `readBarrierMR` proportional to the number of instructions required to access an object. In addition, a 5% of executed bytecodes require write barriers executing the code of Figure 10 for checking illegal accesses and also illegal assignments (see Section 2.5). Thus we estimate the inter-region reference overhead as 0.05 ∗ writeBarrierRegion, where the writeBarrierRegion parameter is the percentage of the average number of instructions executed by the code taged as `writeBarrierMR` proportional to the number of instructions required by an assignment.

Note that read barriers are not strictly necessary. The restriction on critical tasks can be reduced to write barriers checks since reads does not interfere with the GC.[11] Write barriers-supporting MRs include check for both illegal inter-region references (which causes an `IllegalAssignmentError` exception), and illegal access from critical task to objects within the heap (which causes a `MemoryAccesError` exception). We thus add the `getWriteBarrierOverhead()` method to the `MemoryArea` abstract class, which serves to identify region barrier overhead. Note that for write barrier-based collectors (e.g., incremental or generational collectors), this method gives the write barrier overhead caused by the GC.

3.3. Collector Barrier Overhead

Hence, the write barrier cost introduced by the GC is as for regions (i.e., 0.05 ∗ writeBarrierCollector). However, in this case as shown in Figure 11, the writeBarrierCollector parameter is the average cost to preserve the tri-color invariant

```
writeBarrierGC:
   if ((color(X) = black) and (color(Y) = white)) grayObject(Y);
end_writeBarrierGC:
```

Figure 11. Write barrier code for the collector.

(e.g., test if a white object is referenced by a black object, then grey the referenced object and link it to the gray-list) proportional to the number of instructions required by an assignment.

The most common approach to implement write barriers is by in-line code, consisting in generating the instructions executing write barrier events with 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.

4. Minimizing the Write Barrier Overhead

A solution minimizing the write barrier overhead consists in using hardware support such as the picoJava-II microprocessor,[12] which allows performing write barrier checks in parallel with the store operation.

4.1. Using Hardware Support

Upon each instruction execution, the picoJava-II core checks for conditions that cause a trap. From the standpoint of hardware support for GC, the core of 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 assigning a new reference to a field of an object (i.e., when executing `putfield`, `putstatic`, `aputfield_quick`, `aputstatic_quick`, `aastore`, or `aastore_quick` bytecodes). The conditions under which this trap is generated are governed by the values of the `PSR` and the `GC_CONFIG` registers. If the `GCE` bit of the `PSR` register is set, then write barriers are enabled. Hence to disable them it suffices to unset this bit. The `GC_CONFIG` register governs two types of write-barrier mechanisms: page- and reference-based. We can use both mechanisms simultaneously. Also, we can disable either or both of the mechanisms if we do not want to use them (e.g., if the current tracing rate allows disabling the GC). The configuration of this register is summarized in Table 4.

The reference-based write barriers of picoJava-II can be used to implement incremental collectors. An incremental collector traps when a white object is written

Table 4. Garbage collector register (GC_CONFIG).

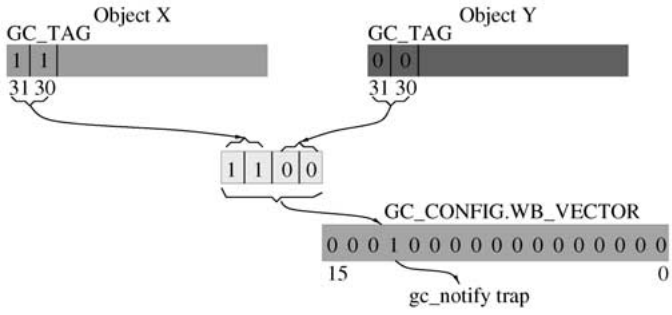
Bits	Field	Type	Description
31 : 21	REGION_MASK	RW	It allows knowing if both, the reference and the stored data belong to the same page.
20 : 16	CAR_MASK	RW	It allows knowing if both, the reference and the stored data belong to the same car.
15 : 0	WB_VECTOR (Write Barrier Vector)	RW	If the corresponding bit is set, then, the above bytecodes signal a gc_notify trap.

into a black object; the GC_TAG field for black objects is %11 and for white objects %00 (see Figure 12).

The page-based barrier mechanism was designed specifically to assist train-based generational collectors (Wilson and Johnstone, 1993). A train-based generational collector traps when, within a larger memory divided into a number of fixed-size spaces, an object (X) references another object (Y) located in the same space but in a different car (see Figure 13).

For example, if in the GC_CONFIG register, we initialize the REGION_MASK field ($\langle 31 : 21 \rangle$ bits) as %00000000000, and the CAR_MASK field ($\langle 20 : 16 \rangle$ bits) as %11111, we divide the memory address space in 32 regions, each one divided in 16 KB cars (see Figure 14). If we choose a %11110 value for the CAR_MASK, then we have 16 regions, and a car size of 32 KB.

We use the page-based write barrier mechanism to detect references across different regions. In order to use this hardware mechanism of picoJava-II, we adapt our algorithm as follows: (i) in the header object, the $\langle 31 : 30 \rangle$ bits give the color of the object and the $\langle 18 : 14 \rangle$ bits give the memory area in which the object is allocated,[13] and (ii) an

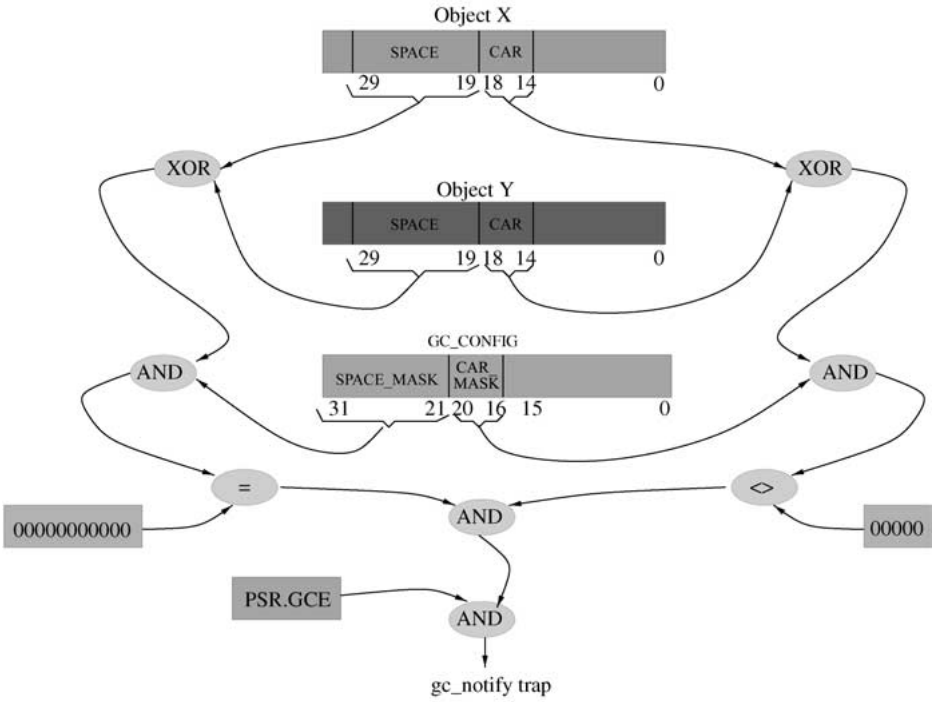


a. Reference-based mechanism.

```
if ( PSR.GCE = 1 ) then
        gc_index <= (X<31:30> << 2) | Y<31:30>
        write_barrier_bit <= (GC_CONFIG >> gc_index) | 0x00000001
        if (write_barrier_bit = 1) then gc_notify trap
```

b. Reference-based pseudocode.

Figure 12. Reference-based write barriers.



a. Page-based mechanism.

```
if ((X<29:19> & GC_CONFIG<31:21>) = (Y<29:19> & GC_CONFIG<31:21>))
      AND ((X<18:14> & GC_CONFIG<20:16>) <> (Y<18:14> & GC_CONFIG<20:16>))
then gc_notify trap
```

b. Reference-based pseudocode.

Figure 13. Page-based write barriers.

associated exception handler determines whether to execute the algorithm described in Section 2.5 to detect erroneous inter-region references, and whether to execute actions preserving the tri-color invariant of the GC. The only overhead is the handling of the exception trap. However, this solution is very costly, due to the high costs of operating system traps.

4.2. Improving Write Barrier Performance

Regarding the proposed solution, which configures picoJava-II to enable page-based and reference-based write barriers, to handle the `gc_notify` trap, 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 immortal region, (B) when it is within the heap, and (C) when it is within a scoped region (see Table 5).

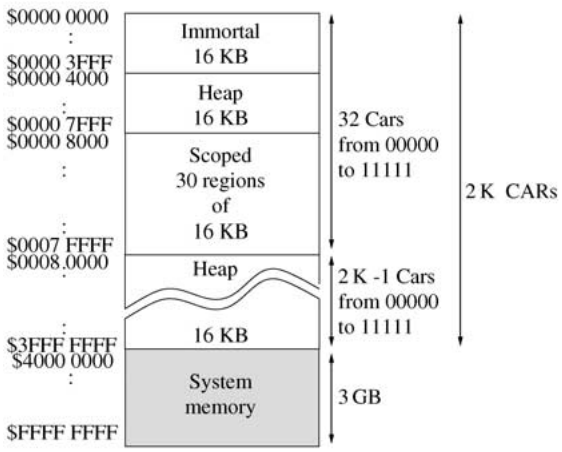


Figure 14. A memory map for memory regions: the $\langle 30:30 \rangle$ bits (i.e., `GC_TAG` field) and $\langle 1:0 \rangle$ bits (i.e., `X` and `H` bits) of the reference object are masked to address the object.

The *A* condition is always allowed and does not require any treatment. For the *B* condition, we must make distinction depending on the color of the object that makes the reference (i.e., the X object):

- B.1 when the X object is black,
- B.2 when the X object is red and the Y object is within the heap.

Finally, for the *C* condition, we make distinction depending on the MR of the X object:

- C.1 is within the immortal region or within the heap,
- C.2 is within a different region than the Y object.

Table 5. Treatments for inter-region references.

X object	Y object	Treatment
Immortal	Immortal	Allowed
Immortal	Heap	Color analysis
Immortal	Scoped	Illegal assignment
Heap	Immortal	Allowed
Heap	Heap	Color analysis
Heap	Scoped	Illegal assignment
Scoped	Immortal	Allowed assignment
Scoped	Heap	Color analysis
Scoped	Scoped	Region stack exploration (when both scoped regions are different)

```
priority_3:
     if (color(X) = black) goto grayObjectY                          // B.1
     if (region(X) = heap) goto memoryAccesError                     // B.2
priority_2:
     if (region(X) <> scoped)
            if  (region(Y) = scoped) goto illegalAssignmentError     // C.1
           if   (region(Y) = scoped) goto checkNetsdRegions          // C.2
     priv_ret_from_trap                                              // A
priority_1:
     goto grayObject                                                 // B.1
```

Figure 15. Treating write barrier exceptions.

Two different mechanisms detect the above conditions: (i) B.1 and B.2 conditions are partly detected by reference-based write barriers and (ii) A, B.1, B.2, C.1, and C.2 conditions are detected by page-based write barriers mechanisms. Then, B.1 and B.2 conditions are detected by both mechanism reference- and page-based write barriers. Since we must treat each condition in a different way, it is pretty interesting to make distinction by hardware whether the trap is caused by a reference- or a paged-based condition[14] (e.g., `gc_notify_0` when reference-based traps and `gc_notify_1` when paged-based traps). In order to improve the performance of critical tasks, the treatment of the B.1 condition is prioritized. Then, we establish three main priority levels, where level 3 is the highest:

1. the reference-based write barrier trap is triggered,

2. the page-based write barrier trap is triggered,

3. both traps are triggered.

The exception with priority 3 treats the B.1 and B.2 conditions. The routine of priority 2 treats the A, C.1, and C.2 conditions. Finally, the routine associated with priority 1 treats the B.1 condition. Note that for the B.2 condition, both write barriers mechanisms are always activated (i.e., the heap have not red objects). These conditions must be treated as follows:

A: is always allowed.

B.1: colors gray referenced object.

B.2: throws the `MemoryAccessError` exception.

C.1: throws the `IllegalAssignmentError` exception.

C.2: explores the region stack associated to the active task.

The above solution minimizes the cost of write barriers, which is zero for intra-region references.[15]

5. Implementation Issues

This section discusses the implementation of a memory management strategy for a Java environment aimed at wireless PDAs. Regarding specifically the offered memory management, it builds upon the RTSJ and the KVM, and integrates the aforementioned solutions for improving the performance of write barriers in GC and region management.

5.1. Integration within the KVM

We are currently implementing the proposed memory management solution within the KVM in a way compliant to the RTSJ. The RTSJ defines the `GarbageCollector` abstract class, which has been specialized through an `IncrementalGC` subclass. We have implemented such a class within the KVM by modifying some files of the interpreter to support our real-time GC (i.e., `garbage.c` to implement the collector algorithm and `interpreter.c` to implement the write barriers, as well as `native.h` and `nativeCore.c` that support the interface for the Java native methods). The incremental collector can be introduced in the system by using the run-time type identification (RTTI) that the class `Class` offers, (e.g., `GarbageCollector gc = (GarbageCollector) Class.forName (' IncrementalGC').`*newInstance*();).

As discussed in the previous section, a significant source of performance improvement for memory management is to exploit hardware aid. We are thus implementing memory management so that it can be run over picoJava-II. In order to make our memory management implementation compliant with this microprocessor, we have modified the object header tag of the KVM as follows: `GC_TAG` $\langle 31:30 \rangle$, `SIZE_H` $\langle 29:19 \rangle$, `CAR_MASK` $\langle 18:14 \rangle$, `SIZE_L` $\langle 13:7 \rangle$, `TYPE` $\langle 6:2 \rangle$, `X` $\langle 1 \rangle$, and `H` $\langle 0 \rangle$. We thus take six bits of the KVM `SIZE` $\langle 31:8 \rangle$ field (i.e., the maximum size of the objects has thus been reduced from 16 MB to 256 KB). Note that the restriction on object size is not severe penalty as suggested by the small average size of Java objects in SPECjvm98 (SPEC, 1998) applications (e.g., Jess 40 Bytes, Db 31, Javac 36, Mtrt 25, and Jack 31). We also reduce the KVM `TYPE` $\langle 7:2 \rangle$ field since 5 bits are sufficient (i.e., only 20 types are handled). These bits have been used as the `GC_TAG` $\langle 31:30 \rangle$, and the `CAR_MASK` $\langle 18:14 \rangle$ fields of picoJava-II (i.e., these fields are used to store the color of the object and the embedding memory region). The KVM `MARK_BIT` that is used by the collector to mark the object is no longer used because objects are marked by color. Then, this bit is exploited to support the `X` bit of picoJava-II. Finally, the KVM `STATIC_BIT` is now used to mean the `H` bit of picoJava-II. Since the collector does not move objects, handles are suppressed. This strategy increases the performance of both, the application and the collector. Then, the `H` bit is fixed to 0, and the core of picoJava-II accesses the object starting a word after the handle. Eliminating handles improves also memory consumption by a word per object. Given the small average size of Java objects the space overhead can be reduced to 12% of

the total dynamic memory space (i.e., $4/(33+4) \times 100$, where 4 and 33 are respectively the handle size and the average size of Java objects in bytes). Since objects in memory regions are not moved, this strategy is always possible even if the objects in the heap are accessed via a handle.

5.2. Experiment

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.[16] 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 MB. Then, the allocation rate is about 1.6 KB/1000-executed bytecodes. The number of garbage collection pass, the total time spent by garbage collection in microsecond, and the percentage overhead introduced by our collector are given in Table 6.

The maximum latency to preempt the incremental collector has been measured as 1 μs. 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 Section 3.2 with SPECjvm98 (SPEC, 1998). And the overhead introduced by the software write barrier test in each assignment, is:

- 31% to maintain the root-set.
- 31% to preserve the tri-color invariant.
- 65% to detect possible illegal references.
- 45% to check a nested scoped level.

Table 6. Garbage collection overhead.

Memory heap	GC pass	Time collecting	Execution time	Percentage overhead
8 MB	51	13.54×10^6	72.87×10^6	18.85
16 MB	27	13.17×10^6	72.72×10^6	18.11
24 MB	17	12.80×10^6	71.99×10^6	17.80
32 MB	13	11.82×10^6	70.50×10^6	16.50

5.3. Additional Considerations

Our solution requires configuring write barriers in picoJava-II. Notice that if the GCE bit of the PSR register is set, write barriers are enabled. The instruction set of picoJava-II provides extended bytecodes allowing access to the PSR and the GC_CONFIG registers (i.e., priv_read_psr, priv_write_psr, priv_read_gc_config, and priv_write_gc_config). The routines given in Figure 16 allow enabling and disabling write barriers. Whereas, Figure 17 shows how reference- and page-based write barriers can be enabled to have our desired configuration. In this example, we have chosen 32 regions with a car size of 16 KB, and we have established the following color codes: %11, %10, %01, and %00 meaning black, gray, red, and white, respectively. The partition of the heap in cars is transparent for the GC by using a mask (e.g., $*FFE0FFFF*$).

This implementation is 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. However, our solution improves the memory management performance, because it minimizes the cost of intra-region references, which has been reduced to the write barrier cost introduced by the GC algorithm in the

```
EnablePageWriteBarrier:                DisablePageWriteBarrier:
    priv_read_psr                          priv_read_psr
    spush 0x1000   //Set GCE              spush 0xEFFF   //Unset GCE
    seti 0x0000                            seti 0xFFFF
    ior                                    iand
    priv_write_psr                         priv_write_psr
    priv_ret_from_trap                     priv_ret_from_trap
```

Figure 16. Enabling and disabling barriers. If the GCE bit of the PSR register is set, then the paged-based write-barriers are able. Hence to disable them it is suffice to set the bit to 0.

```
ConfigureWriteBarrier:
    spush 0x1D00                  //Reference-based barriers
    seti 0x001F                   //Page-based barriers
    priv_write_gc_config
    goto EnablePageWriteBarrier   //Set the GCE bit
```

Figure 17. Configuring write barriers: We have established the following color codes: %11, %10, %01, and %00 mean black, gray, red, and white respectively. And we have divided the address space in 32 regions.

heap, and to zero for the other memory regions. Note further that intra-region references are much more frequent than inter-region references.

6. Conclusion

This paper has presented solutions for improving the performance of memory management in the RTSJ, hence addressing performance improvement of both GC and region management. Our proposal builds upon existing work since the area of memory management in general, and of GC in particular, has for long been deserving a great deal of attention in the programming language and system communities. The contribution of our work comes from the adaptation and integration of relevant solutions, in the context of the RTSJ, based on the analysis of the parameters that are the most influential in memory management performance. In addition, we have discussed the implementation of the resulting memory management solution within the KVM.

We omit write barriers in native code, which may be addressed using either of the two following solutions: (i) forcing the native code to register their writes explicitly, or (ii) 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. Our solutions for improving performance of memory management partly addresses the use of hardware aid by exploiting existing hardware support for Java (i.e., picoJava-II). In general, our study should be complemented with work on improving memory management performance at the hardware level considering both hardware aid and the features of the underlying processor, e.g., impact of garbage collection upon cache management (Kim and Hsu, 2000).

Notes

1. This class supports a region with special memory attributes (e.g., `ALIGNED`, `BYTESWAP`, `DMA`, and `SHARED`).
2. This class supports a memory region with physical properties, having limited lifetime.
3. This method is a member of the `Object` class, which is executed just before discarding an object marked as garbage.
4. Whereas read barriers take actions upon every each object access, write barriers take actions when the application updates pointers.
5. In RTSJ, critical tasks are instances of the `NoHeapRealtimeThread` class, high-priority tasks of the `RealtimeThread` class, and low-priority of the `Thread` class.
6. We can build a `VTMemory` object with a specific GC. Note that in this case, critical tasks must be able to use it.
7. This is always the case for objects within an instance of `LTMemory` but this is not mandatory for objects within an instance of `VTMemory`.
8. Typically, the `ScopedMemory` implementation is made by using `malloc`() and `free`() routines to manipulate memory.
9. In Baker-based algorithms, the work of the GC is performed in small increments triggered by allocations requests.
10. http://www.spec.org/osg/jvm98
11. We apply the same optimization as for the incremental GC which is to use write barriers instead to read barriers.
12. http://www.sun.com/microelectronics/picoJava

13. Note that when the referenced object is from the heap it is not needed accessing the region stack.
14. Actually, the hardware support of picoJava-II does not make distinction, throwing the `gc_notify` for both reference- and page-based, write barriers.
15. Except the cost to maintain the tri-color invariant when using an incremental GC.
16. http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html

References

Ali, K. A. M. 1998. A simple generational real-time garbage collection scheme. *Computing Paradigms and Computational Intelligence (New Generation Computing)* 16(2): 201–221.

Back, G., Tullmann, P., Stoller, L., Hsieh, W. C., and Lepreau, J. 1998. Java operating systems: design an implementation. Technical Report, Department of Computer Science, University of Utah, http://www.cs.utah.edu/projects/flux, August.

Baker, H. G. 1992. The treadmill: Real-time garbage collection without motion sickness. In *Proceedings of the Workshop on Garbage Collection in Object-Oriented Systems, (OOPSLA)*. Also appears as SIGPLAN Notices 27(3): pp. 66–70.

Bollella, G., and Gosling, J. 2000. The real-time specification for Java. *IEEE Computer* June: 47–53.

Cabillic, G., Lesot, J.P., Banâtre, M., Issarny, V., Parain, F., Higuera, T., Chauvel, G., Laserre, S., and Dinverno, D. A flexible distribute Java environment for wireless PDA architectures based on DSP technology. *The Application of Programmable DSPs in Mobile Communications*. John Wiley & Sons, to appear.

Cannarozzi, D. J., Plezbert, M. P., and Cytron, R. K. 2000. Contaminated garbage collection. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, ACM SIGPLAN, May.

Carnahan, L. NIST Requirements Working Group Homepage. Technical Report, http://www.nist.gov/rt-java/carnahan.

Gay, D., and Aiken, A. 1998. Memory Management with Explicit Regions. In *Proceedings of the Conference of Programming Language Design and Implementation (PLDI)*, ACM SIGPLAN, June.

Gay, D., and Steensgaard, B. Stack allocating objects in Java. Technical Report, Research Microsoft, http://www.research.microsoft.com/apl.

Hardin, D.S. 2000. Real-time objects on the bare metal: an efficient hardware realization of the Java virtual machine. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, March.

Higuera, T., Issarny, V., Banâtre M., Cabillic, G., Lesot, J. P., and Parain F. 2000. Java embedded real-time systems: an overview of existing solutions. In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, March.

Higuera, T., Issarny, V., Banâtre M., Cabillic, G., Lesot, J. P., and Parain F. 2000. Region-based memory management for Real-time Java. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, March.

Hilderink, H. 1998. A new Java thread model for concurrent programming of real-time systems. *Real-Time Java*, 30–35 January.

Issarny, V., Banâtre, M., Weis, F., Cabillic, G., Couderc, P., Higuera, T., and Parain, F. 2000. Providing an embedded software environment for wireless PDAs. In *Proceedings of the 9th ACM SIGOPS European Workshop—Beyond the PC: New Challenges for the Operating Systems*, September.

J Consortium, Inc. 1999. Core Real-time extensions for the Java platform. Technical Report, *NewMonics Inc*, http://www.j-consortium.org/rtjwg.html, August.

Kim, J. S., and Hsu, Y. 2000. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the ACM Java Grande 2000 Conference*, June.

de Miguel, M. A. 2000. Solutions to make Java-RMI time predictable. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, March.

Miyoshi, A., Tokuda, H., and Kitayama, T. 1997. Implementation and evaluation of Real-time Java threads. In *Proceedings of IEEE Real-Time Systems Symposium*, December.

Nilsen, K. 1998. Adding real-time capabilities to Java. *Communications of the ACM* 41(6): 49–56, June.

Petit-Bianco, A., and Tromey, T. 1998. Garbage collection for Java in embedded systems. In *Proceedings of IEEE Workshop on Programming Languages for Real-Time Industrial Aplications*, December.

RTJEG (The Real-Time for Java Expert Group). 2000. The real-time specification for Java. Technical Report, *RTSJ*, http://www.rtj.org, June.

SPEC (The Standard Performance Evaluation Council). JVM98 benchmarks. Technical Report, http://www.spec.org/osg/jvm9.

Sun Microsystems. 1999. picoJava-II programmer's reference manual. Technical Report, *Sun Microsystems*, http://www.sun.com/microelectronics/picoJava, March.

Sun Microsystems. 2000. KVM technical specification. Technical Report, *Sun Microsystems*, http://www.sun.com, May.

Sun Microsystems. JavaCard Home Page. Technical Report, *Sun Microsystems*, http://java.sun.com/products/javacard.

Sun Microsystems. EmbeddedJava Home Page. Technical Report, *Sun Microsystems*, http://java.sun.com/products/embeddedjava.

Sun Microsystems. PersonalJava Technology White Paper. Technical Report, *Sun Microsystems*, http://java.sun.com/products/personaljava.

Wilson, P. R., and Johnstone, M. S. 1993. Real-time non-copying garbage collection. ACM OOPSLA Workshop on Garbage Collection and Memory Manageeen, September 1993, ftp://ftp.cs.utexas.edu.

Zorn, B. 1990. Barrier methods for garbage collection. Technical Report CU-CS-494-90, Department of Computer Science, University of Colorado at Boulder, "http://www.cs.colorado.edu, November.

Teresa Higuera obtained a M.S. degree at the Technical University of Madrid and she has just obtained her Ph.D. in computer science from the University of Rennes. Currently, she is at the Complutense University of Madrid, and previously she held a Ph.D. studentship for 3 years at INRIA in Rocquencourt. She worked full-time as teaching assistant at Technical University of Madrid for 7 years and for 2 years as system engineering at IBERIA. Her principal research interests are embedded, real-time and distributed systems. In these fields she has published several papers and technical reports.

Valerie Issarny received her Ph.D. and her ''habilitation a diriger des recherches'' in computer science from the University of Rennes I in 1991 and 1997, respectively. She holds a full-time research scientist position at INRIA since 1993. Her research activity relates to software engineering and distributed systems. In that context, she is leading the Arles research project that examines the construction of mobile distributed systems based on software architecture description. Since 1999, Valerie Issarny is vice-chair of the ACM SIGOPS. She is also chair of the Research Coordination and Training committee of the IST NoE CaberNet on distributed and dependable computing systems.

Michel Banâtre received his ''thèse détat'' degree in 1984. Since 1986 he has a ''Directeur de recherche'' position at INRIA. He is currently leading the ACES (Ambient Computing and Embedded Systems) research group working on embedded systems, Spontaneous Information Systems based on Short Distance Wireless (SDW) technology, context aware services and Java-based operating system for PDA. These research activities are strongly connected with industry partners. He has over 70 publications and patents in the areas of programming languages, distributed systems, fault tolerant architectures, multimedia information systems and context-aware information systems based on SDW technologies.

Jean-Philippe Lesot received his Ph.D. degree in operating systems at the University of Paris 6 in 1999. He is at present a R&D engineer at the INRIA institute. His major work concerns the design and implementation of a flexible Java Virtual Machine. His is jointly specifying a modular development environment for embedded heterogeneous multiprocessor.

Frédéric Parain has just completed his thesis on power aware scheduling for heterogeneous multiprocessor platforms. He is now a research engineer in the INRIA institut and works on a real-time Java distributed processing environment for wireless multiprocessor appliances.

Gilibert Cabillic received his Ph.D. in 1996 from the University of Rennes. He is currently an INRIA researcher (the french national institute for research in computer science and control). His research interests include embedded operating systems, energy management, wireless networks, and heterogeneous computing. He is also working in the area of embedded Java execution environments.