Hardware-based Solution Detecting Illegal References in Real-Time Java[*]

M. Teresa Higuera-Toledano

Facultad Informática, Universidad Complutense de Madrid, 28040 Madrid Spain

Email: mthiguer@dacya.ucm.es

Abstract

The memory model used in the Real-Time Specification for Java (RTSJ) imposes strict assignment rules to or from memory areas preventing the creation of dangling pointers, and thus maintaining the pointer safety of Java. An implementation solution to ensure the checking of these rules before each assignment statement consists to use write barriers executing a stack-based algorithm. This paper provides a hardware-based solution for both write barriers and the stack-based algorithm.

1 Introduction

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. An intermediate approach is to use Memory Regions (MRs) within which both allocation and de-allocation are customized and also the space locality is improved. Application of these two implicit strategies has been studied in the context of Java, which are combined in the Real-time Specification for Java (RTSJ) [3]. RTSJ extends Java to support key features for real-time systems such as real-time scheduling and predictable memory. This paper focuses on how to improve the performance of a Java memory management solution accounting for relevant Java specifications: the RTSJ and the KVM [14] targeting limited-resource and network connected devices, and the picoJava-II [15] microprocessor.

The `MemoryArea` abstract class supports the region paradigm in the RTSJ specification through the three following kinds of regions: (*i*) immortal memory, supported by the `ImmortalMemory` and the `ImmortalPhysicalMemory` classes, that contains objects whose life ends only when the JVM terminates; (*ii*) (nested) scoped memory, supported by the `ScopedMemory` abstract class, that enables grouping objects having well-defined lifetimes; and *(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. When a new scope is entered by calling the `enter()` method of the instance or by starting a new task (i.e., by instancing a `RealtimeThread` or a `NonHeapRealtimeThread`) whose constructors where given a memory region. In the second case, an object created by the task is allocated within memory associated with this scope. When the scope is exited by returning from the `enter()` method, all objects will allocate within the memory associated with the enclosing scope (i.e., the nested outer scope).

Objects allocated within immortal regions live until the end of the application and are never subject to garbage collection. Objects with limited lifetime can be allocated into a scoped region or the heap. Garbage collection within the application heap relies on the (real-time) collector of the JVM. A scoped region gets collected as a whole once it is no longer used. 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 1). Shared scoped regions are used to communication among real-time threads and normal (non-real-time) threads.

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Same, outer, or shared
Local Variable	Yes	Yes	Same, outer, or shared

Table 1. Assignment rules in RTSJ.

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 [3]. In this paper, we specifically treat the issue of dynamic checks for illegal assignment.

[*] Founded by the Ministerio de Ciencia y Tecnología of Spain (CICYT); Grant Number TIC2002-00334.

1.1 Related work

Several researches have examined the possibility of replacing the Java GC by an adequate stack-allocation scheme, which is more predictable. Stack-allocation is desirable because execution time properties are easier to capture than heap allocation. The Tofte-Talpin calculus [17] uses a lexically scoped expression to delimit the lifetime of a region for ML inference systems. Memory for the region is allocated when the control enters into the scope of the region constructor, and is de-allocated 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 type-based analysis, which consists to annotate in the source program every expression creating a value with a region variable.

As the Tofte-Talpin solution, our solution is based on a stack of scoped regions ordered by life-times [10]. Since 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. In [5], we found a region-based approach to memory management in Java using static analysis. As conclusion of this work, fixed-size regions have better performance than variable-sized regions, and region allocation has more predictable and better performance that the GC. But, this solution permits the creation of dangling pointers.

Our solution consists to check the imposed assignment rules preserving dangling pointers dynamically, just when executing the assignment statement. In order to do so, we introduce an extra code in all bytecodes causing an object assignment. This extra code, normally called *write barrier* must be executed before updating the object reference. A similar approach given in [4] 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, which uses also a stack-based memory management that operates dynamically. But, this solution collects memory within the heap, and does not treat another memory region.

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 [19]. This approach is taken in [2] where the implementation uses five runtime heap checks (e.g., CALL, METHOD, NATIVECALL, READ, and WRITE) to ensure that a critical task does not manipulate heap references. Alternatively, our solution instruments the bytecode interpreter, avoiding space problems, but this still requires a complementary solution to handle native code.

The use of write barriers and a stack of scoped regions to detect illegal inter-region assignments has been introduced in [7] and developed in [6]. The solution [7] minimizes the write barrier overhead by using hardware support such as the picoJava-II microprocessor [15], which allows performing write barrier checks in parallel with the store operation. In [8], we compare and evaluate the solutions proposed in [6] and [7]. In [10], we address problems related to the coexistence of MRs and a real-time GC within the heap, such as "*memory access errors*" and "*real-time invariants*" are addressed in detail, as well the region-stack maintenance, and the scoped-region collection.

The aforementioned solutions combine memory regions and a real-time GC collecting objects within the heap. As contrast, in this paper we do-not considerer a real-time GC within the heap, we only address illegal assignments. Then, the solution is simplified. As our major contribution, we introduce a special hardware to support the region-stack algorithm detailed in [9], and the way to improve the performance of checking for illegal assignments to objects within scoped regions. Integrating our specialized hardware support for scoped regions and the write barriers hardware support that picoJava-II provides for generational GCs, both hardware support can work together to detect illegal assignments across regions at negligible time overhead.

1.2 Paper organization

This paper proposes addresses the problem associated with inter-region references introduced by MRs (i.e., illegal assignment). We first present our basic approach, which includes write barriers to detect whether the application attempts to create an illegal assignment, and a description of how scoped regions are supported (Section 2). A hardware-based solution to improve the performance of write barriers, which reduces the write barrier overhead for intra-region assignments to zero, is then given (Section 3). Next, we propose a specialized hardware to support the stack-based algorithm, which makes negligible the time-cost to detect illegal assignments across scoped regions (Section 4). We present basic modifications to the Java VM in order to make it compatible with the existence of memory regions, and evaluate the overhead introduced by write barriers in our solution by instrumenting the KVM (Section 5). Finally, some conclusions and a summary of our contribution; conclude this paper (Section 6).

2 Detecting illegal assignments

A MR implementation must ensure that objects within the heap or the immortal memory cannot reference objects within a scoped region, and objects within a scoped region cannot reference objects within another scoped region that is neither non-outer nor shared. In this section, we use write barriers to detect illegal inter-region assignment at run-time, and a stack-based algorithm to check whether a scoped region is outer to another one.

2.1 Checking for inter-region assignments

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 to be created, 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 (X) within persistent memory (i.e., an outermost region) to another object (Y) to be created.

The header of the object must specify both the region to which the object belongs and the region type. 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. Local variables are also associated with the scope of the active region.

Figure 1 shows the write barrier pseudo-code, which must be introduced in the interpretation of the aforementioned bytecodes. The `region()` function returns: `heap`, `immortal`, or `scoped` depending on the type of the region to which the object parameter belongs, and the `shared()` function returns true when the region to which the object parameter belongs is a shared one. And the `nested(X,Y)` function returns `true`, when the region's scope to which the Y object belongs is the same or outer than the region's scope to which the X object belongs.

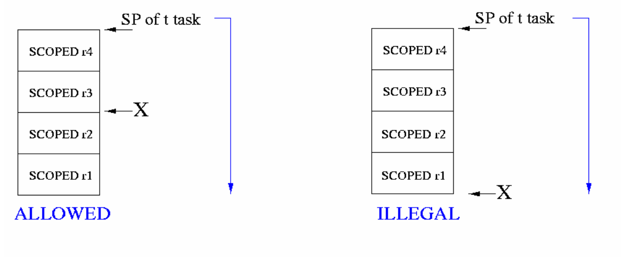
```
wrrite_barrier_code
    if (region(Y)=scoped)
        if (region(X)<>scoped) IllegalAssignment()
        else if ((not shared(Y)) and (not nested(X, Y)))
                IllegalAssignment();
```

Figure 1. Write barrier code detecting illegal assignment.

2.2 Checking nested scoped MRs

In order to detect illegal assignments to scoped regions, every thread has associated a region-stack containing all scoped MRs that the thread can hold. And every scoped region is associated with a reference counter that keeps track of the use of the region by tasks. The MR at the top of the stack is the active region for the task, whereas the MR at the bottom of the stack is the outermost scoped region for the task. The default active region is the heap. When the task does no use any scoped region, the region-stack is empty and the active region is the heap or an immortal MR. Checking nested regions requires two steps.

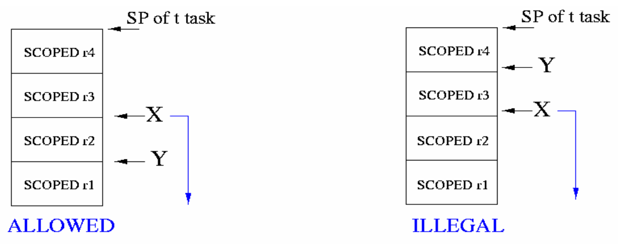
In a first step, the region-stack of the active task is explored, from the top to the bottom, to find the MR to which the X object belongs (see Figure 2). If it is not found, this is notified by throwing a `MemoryAccessError()` exception[1].



a. The X region is found. b. The X region is not found.

Figure 2. First exploration of the region-stack.

A second step explores the region-stack again, considering the start of the search 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 Y object belongs must be outer to the region to which the X object belongs). If the scoped region of Y is found, the `nested(X,Y)` returns `true` (see Figure3).



a. Y is outer to X . b. Y is inner to X.

Figure 3. Second exploration of the region-stack.

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

3 Using hardware support

In this section, we first present an overview of the write barrier hardware support that the picoJava-II microprocessor [15] provides. Next, we introduce a hardware-based solution to improve the write barrier performance of memory regions.

3.1 The picoJava-II write barrier

Upon each instruction execution, the picoJava-II core checks for conditions that may cause a trap. From the standpoint of GC, 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 to an object's field an object reference (i.e., when executing bytecodes requiring write barriers). The conditions that generate the **`gc_notify`** trap are governed by the values of the `GC_CONFIG` and the `PSR` registers. The `GC_CONFIG` register governs two types of write-barrier mechanism: page-based and reference-based. Whereas the reference-based write barriers are used to implement incremental collectors, the page-based barrier mechanism was designed specifically to assist generational collectors based on the *train-based* algorithm [18], which divides the object space into a number of fixed blocks called *cars*, and arranges the cars into disjoint sets (*trains*). This algorithm tracks references across cars within the same train.

3.2 Supporting memory regions

Our solution uses the picoJava-II paged-based mechanism to detect references across different regions by mapping each region in a car. If the `GCE` bit of the `PSR` register is set, then page-based write barriers are enable. The object reference in picoJava-II has 4 fields: `GC_TAG`, `ADDRESS`, `X`, and `H`. The `ADDRESS` field (bits <29:2>) of the reference always points to the location of the object header. In the `GC_CONFIG` register, the `TRAIN_MASK` field (bits <31:21>) allows us to know whether both objects in an assignment X and Y belong to the same train, whereas the `CAR_MASK` field (bits <20:16>) detects whether belong to different cars (see Figure 4).

```
if (PSR.GCE = 1) then
   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
```

Figure 4: Page-based write barrier mechanism.

If, for example, we initialize the `REGION_MASK` field as 0000000000, and the `CAR_MASK` field as 11111, we have only a train divided in 32 cars (regions), each one divided in pages of 16 Kbytes (see Figure 5).

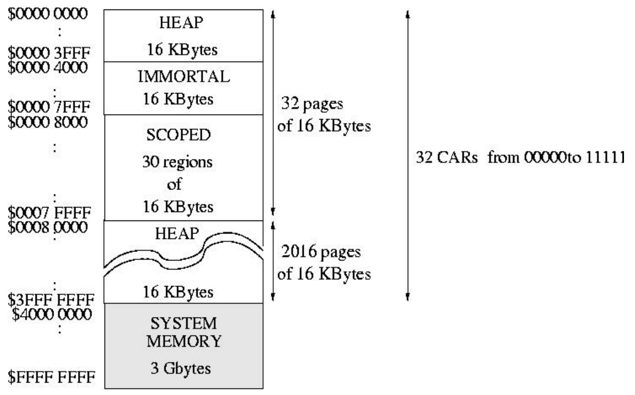


Figure 5: Memory map: `CAR_MASK` with 11111.

The page-based mechanism avoids us to execute the write barrier code when both objects X and Y belongs to the same region (i.e., for intra-region assignments). Then, write barriers are executed only for references across regions (i.e., for inter-regions assignments). Figure 6 shows the associated exception routine to the **gc_notify** trap, which must be executed for inter-region assignments. Note that spatial locality property means that intra-regions assignments are more frequent than inter-region assignments [1]. Then, we improve the performance of our solution at least 50%.

```
gc_notify_trap_code
      if (region(Y) =scoped )
            if (region(X)<>scoped) IllegalAsignment()
            else if ((not shared(Y)) and (not nested(X, Y)))
                        IllegalAsignment();
      priv_ret_form_trap;
```

Figure 6: Treating the **gc_notify** trap.

4 The region-stack algorithm in hardware

In order to improve the performance of our solution, we are interested in reducing the execution time taken to check illegal references caused by assignments among scoped MRs, which depends on the region stack size. In this section we introduce a hardware executing the `nestedRegions(X,Y)` function, which supports the region stack of the active task in an associative memory.

4.1 Checking nested regions

Similarly to the write-barriers mechanism of picoJava-II [15], our proposed hardware checks for the occurrence of *scoped-based* write barriers and notify them by using the **`mr_notify`** trap upon an illegal assignment to an object within a scoped region. The hardware executing the region-stack based algorithm, basically consists in an associative memory called `SCOPED_STACK` (Figure 7).

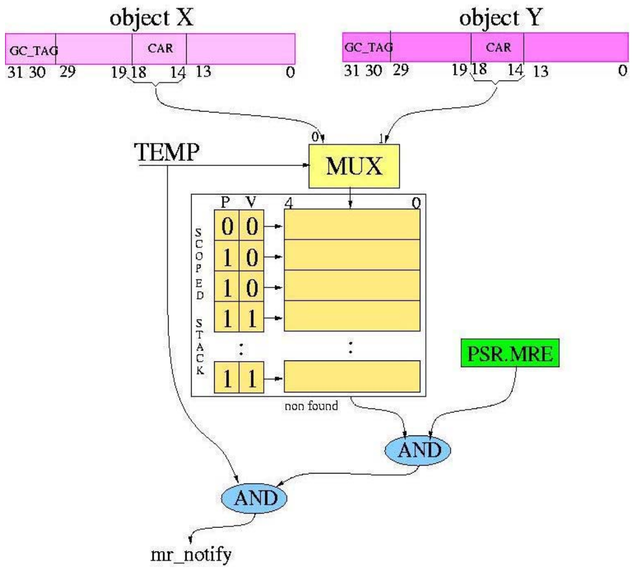


Figure 7: Region-based mechanism for scoped regions.

Each entry of the `SCOPED_STACK` associative memory contains a mark bit, called *Present-Bit* (**P**), which indicates whether the corresponding entry must be considered as an element of the region stack. This bit is used in the first step of the algorithm, when the region of the X object is sought in the region stack. Each entry contains also a valid bit called *Valid-Bit* (**V**), which indicates whether the corresponding entry must be considered at the second step of this algorithm, when the region of the Y object is sought in the region stack.

In the following, we describe the behavior of the two steps of this hardware-based mechanism, which are governed by the **TEMP** signal:

- In a first step (i.e., **TEMP**=0), the region to which the X object belongs is sought in the region stack (i.e., in all entries with **P**=1). If it is found (e.g., at entry *n*), all entries from *n* to the bottom of the stack are validated by setting their **V** bit.

- In a second step (i.e., **TEMP**=1), the region to which the Y object belongs is sought in the sub-stack formed by all entries that have been validated in the previous step (i.e., all entries with **V**=1). If it is not found, the **`mr_notify`** signal activates raising the `IllegalAssignment()` exception.

Note that when the bit **P** is set (**P**=1), the corresponding entry is an element of the region stack of the active task. In contrast, when this bit is unset (**P**=0), the corresponding entry is considered empty. And when the **V** bit is set, the region of the corresponding entry is outer that the region to which the X object belongs (i.e., **V**=1 means that the X object can reference objects within the region). The condition under which the **`mr_notify`** trap is generated can be governed by a *reserved* bit in the `PSR` register (i.e., the <31:23> bits). If this bit (e.g., the <23> bit), which we call *Memory Region Write Barrier Enable* (`MRE`), is set, then scoped-based write barriers are enable. Then, by unseting this bit, we disable the **`mr_notify`** trap.

4.2 Dealing with shared regions

In order to avoid the **`mr_notify`** throwing upon assignments to shared scoped regions, which is an allowed condition, we introduce a new associative memory (see Figure 8): the `SHARED_STACK`, which contains all shared scoped memory regions. The behavior of this mechanism affects only the second step of the algorithm. In this case, the Y object is soughed in both associative memories (i.e., the `SCOPED_STACK` and the `SHARED_STACK`), whether it is not found in neither of the associative memories and the PSR.MRE bit is set (i.e., memory region write barriers are allowed), the **`mr_notify`** signal traps.

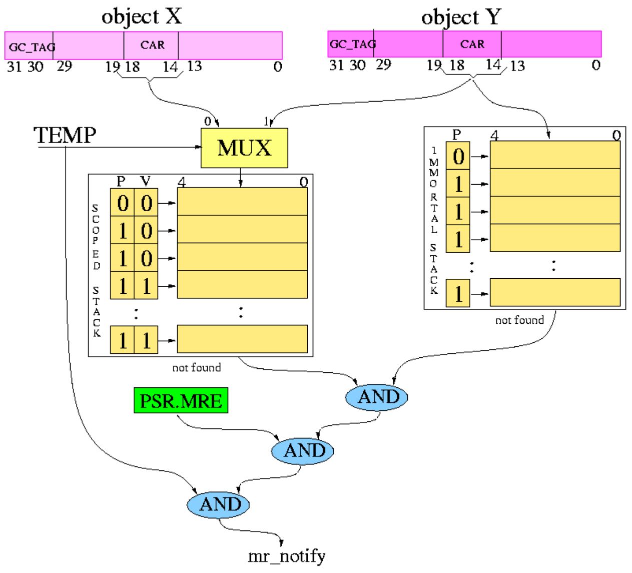


Figure 8: Region-based mechanism for shared regions.

We consider that the <31:30> bits of the object reference are used to specify the type of the region to which the object belongs (e.g., 00 for the heap, 01 for immortal, 10 for scoped non-shared, and 11 for scoped shared). Figure 9 shows the code of the associated exception routine to the **gc_notify** trap. This code, first checks for the region to which the Y object belongs, and whether it is scoped non-shared enables the introduced hardware-based write barrier mechanism.

```
gc_notify_trap_code
            if  (region(Y) =scoped )
                        if  (region(X)<>scoped) IllegalAsignment()
                        else  if (not shared(Y))  begin
                                    enable_mr_notify;
                                    nop;
                                    disable_mr_notify
                        end;
            priv_ret_form_trap
```

Figure 9: Code for the **gc_notify** trap.

Each time a task is scheduled for execution, all entries of both memories the `SCOPED_STACK` and the `SHARED_STACK` must be configured respectively with the scoped region stack and the shared scoped MRs that the scheduled task can access. This configuration introduces some overhead at context-switch time. In order to access/configure the `SCOPED_STACK` (`SHARED_STACK`) memory, we extend the picoJava-II instruction set by introducing the priv_read_scoped_stack (priv_read_shared_stack) and priv_write_scoped_stack (priv_write_shared_stack) bytecodes, which operands are the entry index and a memory address to load/store the region identifier.

5 Integration within the KVM

We have modified the KVM [14] to implement three types of memory regions: *(i)* the heap that is collected by the KVM GC, *(ii)* immortal that is never collected and can not be nested, and *(iii)* scoped that have limited live-time and can be nested. These regions are supported by the `HeapMemory`, the `ImmortalMemory`, and the `ScopedMemory` classes. Unlike RTSJ, in our prototype the `ScopedMemory` class is a non-abstract class, and the `ImmortalPhysicalMemory` class has not been implemented. We have limited to 32 the number of regions. We consider that regions are paged, and the page size is 16 Kbytes. Also, the maximum number of pages that a region can hold is has been limited to 8. We consider further that a maximum of 64 tasks can reference objects in the same scoped MR.

5.1 Memory footprint

We maintain a *region-structure* of 2 words for each MR object in the system with the following format: `REGION_TYPE` <31:30>, `REGION_ID` <29:25>, `OUTER_REGION_ID` <24:20>, `REFERENCE_COUNTER` <19:14>, `INITIAL_SIZE` <13:11>, `MAXIMUN_SIZE` <10:8>, and `USED_PAGES` <7:0>. Where the `REFERENCE_COUNTER`, the `INITIAL_SIZE`, and `MAXIMUN_SIZE` fields allow us to know respectively: the number of tasks that can allocate or reference objects in the region, the initial number of pages of the region, and the number of pages that the region can hold. The `USED_PAGES` field is a bitmap indicating the pages of the region. If for example the `USED_PAGES` field contains the 00000101 value that means that pages 0 and 2 of memory compounds the region (i.e., if the `REGION_ID` contains the 00000 value, pages 0 and 32 of memory compounds the region). The region-structure increases the memory footprint as maximum of 128 Bytes. Note that these region-structures forms a *scope-tree* [12] where the heap is the root and immortal regions are not included.

In order to adapt the KVM objects to the picoJava-II microprocessor, we add a word to the object header of the KVM. The added word includes the following fields: `REGION_TYPE` <31:30> (`GC_TAG` in picoJava-II) and `REGION_ID` <18:14> (`CAR_ADDRESS` in picoJava-II). Where the `REGION_ID` field specifies the MR to which the object belongs, and the `REGION_TYPE` specifies the region type (e.g., 00 for the heap, 01 for immortal, 10 for scoped non-shared, and 11 for scoped shared). 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 identification and type of the region to which the object belongs (i.e., `REGION_TYPE` <31:30>, `SIZE_H` <29:19>, `REGION_ID` <18:14>, `SIZE_L` <13:8>, `TYPE` <7:2>, `MARK_BIT` <1>, and `STATIC_BIT` <0>). Note that the maximum size of the object has been reduced from 16 Mbytes to 512 Kbytes; given the small average object size that the specJVM [16] applications present (i.e., about 32 Bytes), we optimize for small objects.

5.2 Worst case for write barrier

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 that **n** is the maximum bound of executed evaluations to check whether a region is outer than another one.

We have limited the number of scoped nested levels to 16, which allows us to support the region stack of each task in 5 words. Then, the introduced memory footprint is not important. The maximum write barrier cost introduced per assignment is when exploring the 16 levels of the region stack. Note that this cost includes both the evaluations besides chasing the stack and the evaluations to explore the stack (i.e., (3+**n**); **n**<=16). With this implementation, the overhead introduced in the KVM to evaluate a condition of the write barrier test is about 17% per assignment. This means that when introducing a condition in the interpretation of a bytecode causing write barriers, we increase 17% the execution time of the bytecode, and the maximum write barrier overhead per assignment is 323%.

Since the introduced hardware minimizes the time cost required to explore the stack to the time that picoJava-II spends to catch a trap, we estimate the maximum write barrier cost introduced per assignment as 50%.

5.4 Average write barrier overhead

To obtain the average write barrier overhead, two measures are combined: the number of events, and the cost of the event. We use an artificial collector benchmark which is an adaptation made by Hans Boehm from the John Ellis and Kodak benchmark[2]. 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:` $19*10^6$, 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 with SPECjvm98 in [11].

Considering that all the objects in the system have the same probability to be accessed, let h, i and s be respectively the proportion of objects within the heap, an immortal region, or a scoped region. Then, we compute the average write barrier cost introduced per assignment as: $0.17((h+i+\mathbf{s})+s(h+\mathbf{s}+i)+s*s(1+\mathbf{n}/2))$. Recall that to estimate the average write barrier overhead we must consider the number of events (i.e., 5%). Since $h+i+\mathbf{s}=1$ and **n**<=16, we estimate that our instrumented KVM runs slowdown as average $0.05*0.17(1+\mathbf{s}+\mathbf{9s}^2)$, and at the most 16.15%.

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

Let further α and ξ denote respectively the percentage of inter-region and intra-regions references, found in the assignments made by a task. By using the picoJava-II page-based write barrier support, the introduced overhead is factorized by α; α<=0.5. Finally, by using the specialized hardware support for the region-stack algorithm, we estimate the average write barrier overhead as $0.05*\alpha*0.17(1+\mathbf{s}+\mathbf{s}^2)$, and its bound as 1.275%.

5.5 Configuring write barriers

Considering 32 regions and a page size of 16 Kbytes, we introduce: *(i)* a routine to configure the `TRAIN_MASK` field with the `00000000000` value and the `CAR_MASK` field with the `11111` value (see Figure 10), *(ii)* a routine to enable and disable page-based write barriers (see Figure 11), and *(iii)* a routine to enable/disable this mechanism (see Figure 12).

```
configure_page_based_WB
     priv_read_gc_config      // read the GC_CONFIG register
     spush 0x001F             // TRAIN_MASK=00000000000
     seti 0xFFFF              // governs referenced-based WB
     iand                     // and(GC_CONFIG, 0x001FFFFF)
     spush 0x001F             // CAR_MASK=11111
     seti 0x00000             // governs referenced-based WB
     ior                      // or(GC_CONFIG, 0x001FFFFF)
     priv_write_gc_config     // write the GC_CONFIG register
```

Figure 10: Configuring page-based write barriers.

```
enable_gc_notify                  disable_gc_notify
     priv_read_psr                     priv_read_psr
     spush   0x10000                   spush   0xEFFF
     seti   0x0000                     seti   0xFFFF
     iand    // set bit GCE            iand    // unset bit GCE
     priv_write_psr                    priv_write_psr
     ret                               ret
```

Figure 11: Enabling/disabling page-based write barriers.

```
enable_mr_notify                  disable_mr_notify
     priv_read_psr                     priv_read_psr
     spush   0x0080                    spush   0xFF7F
     seti   0x0000                     seti   0xFFFF
     ior            //set bit 23       iand          //unset bit 23
     priv_write_psr                    priv_w rite_psr
```

Figure 12: Enabling and disabling region-based barriers.

6 Conclusions

The RTSJ specification imposes restricted assignments rules that keep longer-lived objects from referencing object in scoped memory, which are possibly shorter live. In our solution, the detection of illegal assignments related with memory regions, is made dynamically by introducing a write barrier mechanism based on a region-stack associated to the active task.

Some critics to RTJS consider that several aspects of the specification have not been resolved yet [12], and that drawbacks are significant when considering memory management [13]. Regarding our software-based solution, we found only two problems: the high overhead that introduces the dynamic check of illegal assignment for scoped MRs, and that this overhead must be bounded by limiting the nested scoped levels. Our solution, to improve the performance of memory management, partly addresses the use of hardware aid by exploiting existing hardware support for Java (i.e., the picoJava-II microprocessor). The performance of this solution has been reduced nearly zero by using specialized hardware, which also avoids us to limit the nested scoped levels.

Our hardware-based solution is efficient, but not very flexible, because we must configure the system to determine the virtual region memory map, which can be unpractical for classes dealing with I/O mapped memory (e.g., `ImmortalPhysicalMemory`). Also requires the size of a region to be multiple of the car size, which may introduce internal fragmentation. These problems can be avoided by using the header of the object in the write barrier mechanism instead of the object reference. Another problem with our solutions is that we omit write barriers in native code, which may be solved by forcing the native code to register their writes explicitly.

References

[1] H.G. Baker. "Infant Mortality and Generational Garbage Collection". In Proc. of the Workshop on Garbage Collection in Object-Oriented Systems. OOPSLA'91. ACM SIGPLAN Notices 1993.

[2] W.S. Beebe and M. Rinard. “An Implementation of Scoped Memory for Real-Time Java”. In Proc of 1st International Workshop of Embedded Software (EMSOFT), 2001.

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

[4] 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.

[5] M. Christiansen, P. Velschow. “Region-Based Memory Management in Java”. Master’s thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

[6] M.T. Higuera, V. Issarny, M. Banatre, 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, to be published.

[7] M.T. Higuera, V. Issarny, M. Banatre, 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 2001.

[8] M.T. Higuera and, V. Issarny “Analyzing the Performance of Memory Management in RTSJ”. In Proc. of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2002.

[9] M.T. Higuera. "Memory Management Solutions for Real-time Java". PHD Thesis. INRIA. March 2002.

[10] 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 2002.

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

[12] K. Nielsen. “Real-Time Programming with Java Technologies”. In Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2001.

[13] K. Nilsson and T.Ekman. “Deterministic Java in Tiny Embedded Systems”. In Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2001.

[14] Sun Microsystems. “KVM Technical Specification". Technical Report. Java Community Process, May 2000. http://java.sun.com.

[15] Sun Microsystems. “picoJava-II Programmer’s Reference Manual". Technical Report. Java Community Process, May 2000. http://java.sun.com.

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

[17] 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.

[18] Wilson P.R. and Johnston M.S. “Real-Time Non-Copying Garbage Collection”. ACM OOPSLA Workshop on Garbage Collection and Memory Management. September 1993.

[19] Zorn B. “Barrier Methods for Garbage Collection”. Technical Report CU.CS. Department of Computer Science. University of Colorado at Boulder. http://www.cs.colorado.edu. November 1990.

However, we think that for some kind of real-time systems like embedded or critical ones, the memory management model can be simplified as in [14].