Towards an Analysis of Garbage Collection Techniques for Embedded Real-Time Java Systems *

M. Teresa Higuera-Toledano
DACYA, Facultad de Informática, Universidad Complutense de Madrid,
Ciudad Universitaria, Madrid 28040, Spain
Email: mthiguer@dacya.ucm.es

Abstract

From a real-time perspective, the Garbage Collector *(GC) introduces unpredictable pauses that are not tolerated by real-time tasks. Real-time collectors eliminate this problem but introduce a high overhead. Another approach is to use Memory Regions (MR) within which allocation and deallocation are customized. This facility is supported by the memory model of the Real-Time Specification for Java (RTSJ). This paper provides an indepth analytical investigation of the problems and solutions of Java garbage collection techniques regarding it use in embedded real-time systems.*

keywords: Java, embedded, real-time, write barriers, memory management, garbage collection

1 Introduction

Real-time programs must execute with time constraints, which are part of the system's behavioural requirements [2]. Many aspects of modern general-purpose computers have soft real-time execution issues (e.g., the software of the monitor mouse activities). Then, real-time programs must not only generate correct results; those results must further be generated *on time*. In addition, these results, in an embedded system, include control signals (e.g., sensors readings and actuator reactions). Examples of embedded real-time systems are flight control systems, automated manufacturing plants and telecommunications and control systems. Normally these programs operate with bounded resources, including CPU time and memory. Embedded real-time programs must be reliable, predictable, and deterministic even when the environment changes.

* This research was supported by Consejera de Educacin de Comunidad de Madrid, Fondo Europeo de Desarrollo Regional (FEDER) and Fondo Social Europeo (FSE), through BIOGRIDNET Research Program S-0505/TIC/000101, and by Ministerio de Educacin y Ciencia, through the research grant TIC2003-01321.

The Java environment provides attributes that make it a powerful platform to develop embedded real-time applications (e.g., architecture-neutral, multithreaded, dynamic loading, and garbage collection). However, it does not provide predictability facilities nor bounded resource usage, which are needed for the above applications. In general, Java presents some problems regarding its use in embedded real-time environments. The *National Institute of Standards and Technology* (NIST) has produced a basic requirements document for a standard real-time Java API extension[1]. Where the memory management is one of the major issues that need research when considering the extension of Java for real-time.

In this paper, we present an analysis of the problems that Java garbage collection techniques present regarding their use in embedded real-time systems, and how these problems has been resolved in some Java solutions. We study the memory fragmentation problem as consequence of a conservative scanning (Section 2). We analyze the simple garbage collection technique that makes of memory a critical region, and the way to make it incremental (Section 3). We study the incompatibility of hard real-time (critical) systems with pre-emption latencies of the garbage collector (Section 4). We give comparison of the studied solutions (Section 5). Finally a summary of our contribution conclude this paper (Section 6).

2 Conservative scanning

Real-time garbage collection must assure memory availability for newly created objects without interfering with the real-time constraints. An important source of unpredictability in Java is the GC. The JVM specification does not specify how objects should be represented in the heap, and the GC technique employed in Java depends on the virtual machine implementation. The Sun JDK [8] and SDK [9] use a mark-and-sweep GC [7] that compacts the heap, avoiding fragmentation problems. This collector is conservative with

Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE

respect to the native stack, but accurate with respect to the heap and the Java stack. A conservative collector can introduce memory leaks when an apparent reference is not a real reference (e.g., when an integer is treated as a pointer to an object). The strategy adopting by these JVMs relies on occasionally running a compacting GC which implies some degradation of real-time guarantees. To reduce the cost of object relocation, each object has a non-moving handle that points to the location of the object header (see Figure 1).

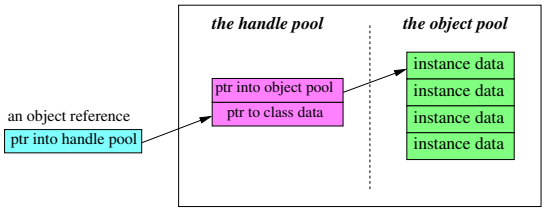

Figure 1. Object format with handle.

When an object is relocated, its handle is updated. In this way, relocating objects is transparent to the application program, which always accesses objects using their non-moving handle. The heap is organized in two sections: the handle space and the object space. Since the handle space consists of fixed size objects (i.e., two words of 32 bits, the first word provides a pointer to the object and the second, a pointer to the object's class), it does not need be compacted. If the piece is not free, the data of an object follows the header word. Compactation is made in two phases: the object space is first compacted, and the handles are then updated.

The HotSpot JVM provides an accurate GC, which eliminates object handles (see Figure 2). Then, the collector must find and update all references to an object when the object is relocated. Indirect handles make relocating objects easier, but it introduces performance degradation because references to instance variables require two memory accesses. Eliminating handles improves also memory consumption in a word per object (approximately 8% of the total Java heap space).

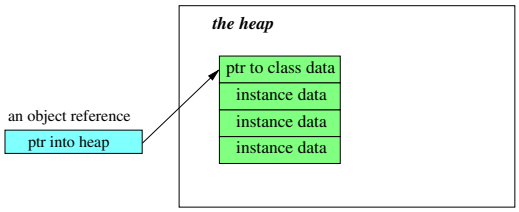

Figure 2. Object format without handle.

The Sun KVM [10], supports accurate garbage collection. For this purpose, the object header has been augmented with a word tag that stores information about the type of the object and the object size (i.e., $\texttt{SIZE} < 31 : 8 >$, $\texttt{TYPE} < 2 : 7 >$, $\texttt{MARK\_BIT} < 1 >$, and $\texttt{STATIC\_BIT} < 0 >$). The collector utilizes the information of the `TYPE` field to perform a different scanning depending on the type of the object. Since this collector does not move objects, handles are suppressed (see Figure 3). This strategy increases the performance of both the application and the collector.

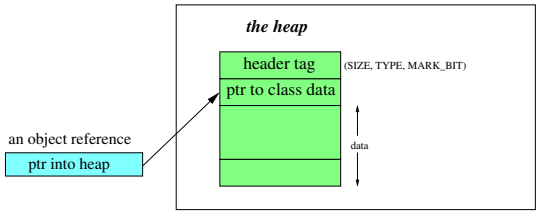

Figure 3. Object format in KVM.

In [11] D.F. Bacon presents a real-time GC algorithm implemented in the Jikes RVM, which uses a simple segregated free-list approach. This solution is based on the *locality size* property, i.e., the objects sizes allocated frequently in the past will tend to have a high correlation with objects sizes allocated in the future. This property allows blocks in a particular size class to be re-used. That means that when the GC will find unused blocks they are inserted in the segregated free-list. Since object allocation can cause external fragmentation, this collector performs defragmentation.

3 Simple garbage collection

The GC of standard Java runs as a low-priority background thread. If the CPU is idle because there are I/O operations, there is no problem. On the other hand, if some threads need memory, the GC is scheduled for execution, leading to suspend all the other threads. Also the GC runs when there is no memory space left to handle application requests, and through the explicit invocation of `System.gc()`. Once the GC is started, it must be executed until completion. It can-not be stopped or preempted, because this would leave the heap in an inconsistent state.

In real-time systems, the GC cannot halt the program and work as one atomic action, then small garbage collection units (called *increments*) must be interleaved with the program execution, which does slow down the application processing. Therefore, the goal for concurrent collectors is keeping their pause times low, while simultaneously increasing the application throughput. In order to synchronize the concurrent execution of the application and the collector, there are two basic techniques called *read barriers* or *write barriers*. A read/write barrier traps pointer loads/stores from/to the heap and records some information that prevents race carrier conditions between the application and the collector.

Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE

The HotSpot JVM [12] provides a generational collector based on the train algorithm [13], which provides constant pause times by dividing the collection of the old-space into many tiny steps. This makes possible to run this collector with multimedia applications. But it is not adequate for hard real-time applications. It is not good, because the guaranteed upper limit on pause times is too large. The pauses introduced by this collector provides constant pause times (i.e., typically less than $10\ Milliseconds$).

The real-time collector of the Jikes RVM [11] use read barriers; this is a copy-based collector. In order to maintain a to-space invariant in the mutator, each object contains a forwarding pointer that normally points to itself, but when the object has been moved, points to the moved object. In this solution the stack processing is not incremental yet. For each pointer, read or write (depending on the barrier strategy), the compiler must generate the necessary additional instructions in the application object code. Write barrier techniques are cheaper and more predictable than read barriers [18].

Alternatively, read and write barriers can be assisted by specialized hardware. The picoJava-II [14] microprocessor specification executes most of the JVM instruction set directly in hardware. The core of this microprocessor checks for the occurrence of write barriers, and notifies them using the `gc_notify` trap. The reference-based write barriers of picoJava-II can be used to implement incremental collectors based in the tri-color algorithm [15], whereas the page-based barrier mechanism of was designed specifically to assist generational collectors based on the train algorithm [16]. Both write barrier mechanisms allow us to improve the performance of both the collector and the application by disabling write barriers execution when disabling the collector. The picoJava architecture description has not been implemented.

The Jikes RVM [11] uses two different scheduling politicizes: one based on time, the other based on work. On time-based scheduling, the execution of the collector and the mutator is interleaved using a fixed time quanta. Here there are two fundamental parameters: the *mutator quantum* that is the amount of time that the mutator can run before the collector can execute, and the *collector quantum* that is the time based collector quantum. In the work-based collector the collector and the mutator interleave their execution based on fixed amounts of allocation and collection. In this case the two fundamental parameters are: the *work-based mutator quantum* that is the amount of memory that the mutator can allocate before the collector can run, and the *work-based collector quantum* that is the amount of memory that the collector can allocate before the mutator yields to it. Both schedulers are good for high priority tasks, but for critical tasks they presents latency problems.

4 The priority inversion problem

The priority inversion problem arises when two tasks τ_i and τ_k with different priorities ($\tau_i < \tau_k$) attempt to access shared data. If the task τ_k, with higher priority, gains access first, the proper priority order is maintained. On the other hand, if the task that gains access is the task τ_i, the task τ_k can be blocked if it requests access to the shared data before task τ_i exits. It may further occur that a third task τ_j with intermediate priority (i.e., $\tau_i < \tau_j < \tau_k$) arrives in this situation. Then, τ_i will be resumed, and task τ_k is blocked during execution of task τ_j also. This blocking period can be arbitrarily long. In order to avoid that critical tasks lost deadlines because the collector, RTSJ makes distinction between three main kinds of tasks:

1. *Low-priority tasks:* are tolerant with the GC.
2. *High-priority tasks:* cannot tolerate unbounded preemption latencies.
3. *Critical tasks:* cannot tolerate preemption latencies.

Whereas high-priority tasks require a real-time GC, critical tasks must not be affected by the GC, and as a consequence cannot access any object within the heap [3]. A reference of a critical task to an object allocated in the heap causes the `MemoryAccessError()` exception. RTSJ [3] defines memory allocation and reclamation specifications that enable the use of real-time compliant garbage collection algorithms without prescribing any specific solution to the technique employed by the GC within the heap. As a complement and alternative to the (real-time) GC, RTSJ provides also an interesting region-based memory allocation technique, which enables grouping related objects within a region.

The Real-time Core Extension for the Java Platform [4] solution has two separate heaps, one for common threads, and the other for core tasks. The core objects are not relocated and garbage collected, and core methods do not include code for synchronization with the GC. Since tasks and threads may share objects, memory sharing must be supported by the underlying RTOS. To make an object eligible by the GC, the method `unanchor()` must be invoked. This is an intermediate solution between explicit memory deallocation and garbage collection.

5 Comparison

To evaluate the GC of studied JVM implementations, we use several important criteria: memory requirements and speed execution of the GC *(efficiency)*, the degree to which it is possible to know memory and CPU time requirements of the application *(predictability)*, how quickly the GC can

Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE

be preempted by the application *(latency)*, the difficulty of integrating the GC in a possible implementation of the Java specifications *(integration)*, and the adaptation degree to a tiny embedded control device *(embedded)*. A comparison is summarized in Table 1 in terms of their strengths and weaknesses. We use **A**, **M**, and **-** to mean the extent to which the corresponding issue is addressed: in detail, only partly addressed, and not addressed.

Feature	Efficiency		Predictability		Latency	Integration	Embedded
JVM	CPU	Mem	CPU	Mem			
KVM	M	A	-	A	-	-	A
picoJava-II	A	-	A	-	M	A	A
JDK/SDK	M	-	-	-	-	M	-
HotSpot	A	M	M	A	M	M	-
Jikes RVM	A	A	M	M	M	M	-
RTSJ	-	-	M	A	A	A	A

Table 1. Comparison of Studied Solutions.

6 Conclusions

Garbage collection simplifies programming, but is often considered infeasible in real-time systems. The problem with GC algorithms is that to guarantee sufficiently short delays, we introduce large overheads (e.g., by introducing write barriers). With current hardware and sophisticated reclamation techniques, this is no longer true. Thus, for a large class of interactive applications (e.g., multimedia), there is no need to exclude automatic memory management. However if hard-real-time demands are imposed, there are problems with the GC latency, which can result in too much jitter in some high real-time tasks, or in missing deadlines.

In RTSJ, the way to offer real-time guarantees is by turning off the GC during the execution of critical tasks, which only allocates objects in memory regions and cannot reference objects within the heap. Some real-time tasks can allocate and reference objects within the heap, whereas others (critical) are not allowed to allocate nor reference objects within the heap.

References

[1] L. Carnahan and M. Ruark, Requirements For Real-time Extensions For the Java Platform http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf

[2] J. Mathai, Real-time Systems Specification, Verification and Analysis, Prentice Hall, 1996

[3] The Real-Time for Java Expert Group, Real-Time Specification for Java, http://www.rtj.org

[4] "J Consortium, Inc, Core Real-Time Extensions for the Java Platform, *NewMonics, Inc*, http://www.j-consortium.org

[5] H.G. Baker, The Treadmill: Real-Time Garbage Collection without Motion Sickness, OOPSLA, 1991

[6] D. Gay and A. Aiken, Memory Management with Explicit Regions, ACM SIGPLAN PLDI, 1998

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

[8] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1997

[9] B. Venners, Inside the Java 2 Virtual Machine, McGraw-Hill, 2000

[10] Sun Microsystems, KVM Technical Specification, Java Community Process, 2000

[11] D.F. Bacon, P.Cheng and V.T. Rajan A Real-Time Garbage Collector with Low Overhead and Consistent Utilization, ACM-POPL, 2003

[12] Sun Microsystems, The Java HotSpot Virtual Machine, v1.4.1, http://java.sun.com/products/hotspot 2002

[13] J. Seligman and S. Grarup, Incremental Mature Garbage Collection Using the Train Algorithm, LNCS Springer-Verlag, 1995

[14] Sun Microsystems, picoJava-II Microarchitecture Guide, http://www.sun.com/microelectronics/picoJava, 1999

[15] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholtenand, and E.F.M. Steffens On-the-fly Garbage Collection: An Exercise in Cooperation, Communications of the ACM, 1978

[16] R.L. Hudson and R. Morrison and J.E.B. Moss and D.S. Munro, Garbage Collecting the World: One Car at a Time, ACM SIGPLAN, 32, 19, 1977

[17] L. Sha and R. Rajkumar and J.P.Lechoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization, IEEE Transactions on Computers, 39, 9, 1990

[18] S.M. Blackburn and A.L. Hosking, Barriers: Friend or Foe?, International Symposium on Memory Management, ACM-ISMM, 2004

Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE