The Impact of Realtime Garbage Collection on Realtime Java Programming

Dr. Fridtjof Siebert
siebert@aicas.com
Director of Development
aicas GmbH
Haid-und-Neu-Straße 18
76131 Karlsruhe, Germany
www.aicas.com

September 28, 2005

Abstract

Extensions like the Real-Time Specification for Java (RTSJ) enable the use of Java in more and more time-critical application domains. The RTSJ enables the development of realtime code in Java even though a classical garbage collector causes unpredictable pauses to non-realtime code.

This presentation gives an overview of how a modern realtime garbage collectors operates. It presents the impact this has on the developer of complex applications that need to perform time-critical and non-time-critical tasks. The use of realtime garbage collection technology simplifies the application development even in systems that do not use dynamic memory allocation within realtime code.

1 Introduction

The enormous success of Java technology is due to the many advantages, such as higher productivity and safety, the language brings to the developer. Even critical applications, as in automotive or aerospace control, can profit from these advantages [17, 1].

Language extensions like the Realtime Specification for Java [5] have made it possible to use Java implementations even though a garbage collector may interrupt the execution of normal Java code in an unpredictable way. New specific features such as realtime threads that cannot access the garbage collected heap make it possible to develop code that has predictable timing behavior and that can be used for hard realtime tasks.

In parallel to the definition of realtime extension to the Java environment such as the Realtime Specification for Java, the technology for deterministic automatic memory management has improved significantly. Realtime garbage collectors that do not require to make any restrictions on the type of memory management that is used for threads that require realtime behavior.

Realtime applications have to perform a set of tasks with different requirements on their timing behavior. Typically, these applications have only few tasks that require hard realtime execution with strict deadlines, while most of the rest of the application can be considered soft realtime. Furthermore, the hard realtime tasks are typically simple enough such that complex features such as dynamic memory allocation are not considered needed for these tasks. This insight has led to the definition of specific memory areas with simpler memory reclamation rules that are not under the control of the garbage collector.

Even though hard realtime tasks do not require dynamic features, these tasks typically need to communicate with other tasks that are less time critical. Typical means for this communication are the use of shared memory and synchronization via Java monitors.

This sharing of resources with non-realtime or lower-priority tasks turns out to be a major drawback of schemes that use separate thread classes and separate memory areas for realtime and non-realtime tasks. In a system that is split into a realtime part that only accesses specific memory areas that are not accessible by normal tasks that may be interrupted by the garbage collector, and a non-realtime part that can make full use of dynamic features and garbage collection, communication between these to parts is becomes very difficult. A realtime task that needs access to a monitor that may as well be held by a non-realtime task that may be stopped by garbage collection activity will suffer from garbage collection pauses exactly as the non-realtime task does.

Modern hard realtime garbage collection techniques can be employed to solve this problem: A system that uses deterministic garbage collection that occurs at predictable times and that is interruptible after very small incremental steps permits hard realtime tasks to access the same memory as non-realtime tasks and permits the use of common synchronization mechanisms. Even though the hard realtime tasks itself may not require any dynamic features such as dynamic memory management, the use of realtime garbage collection will reduce the complexity of the overall system significantly. It will allow straightforward communication between the realtime and non-realtime parts of the application via shared memory and shared monitors without introducing the risks of blocking hard realtime tasks for garbage collection activity.

The rest of this paper is organized as follows. Section 2 will give a short overview of the Realtime Specification for Java. The hard realtime garbage collector developed for the JamaicaVM realtime Java implementation will then be presented in Section 3. The effects of the use of such a garbage collector on the development of realtime software will be illustrated in Section 4. Section 5 will present recent work on realtime garbage collection technology, section 6 concludes this paper.

2 Realtime Specification for Java

The aim of the Realtime Specification for Java (RTSJ) [5] is to extend the Java language definition and the Java standard libraries to support realtime threads, i.e., threads whose execution conforms to certain timing constraints. Nevertheless, the specification is compatible with different Java environments and backwards compatible with existing, non realtime Java applications.

One of the biggest advantages of Java technology, the safe memory management through automatic garbage collection, is also the biggest problem for the use in realtime systems. However, the garbage collector is the basis for the safety mechanisms and is consequently compulsory.

The garbage collector in a classic Java system can stop all application threads at an unpredictable time and for an unpredictable duration. This leads to pauses as illustrated in Figure 1. These pauses make the prediction of timing behavior of the application impossible. To permit realtime programming, one therefore needs a means to avoid these pauses: a deterministic incremental garbage collector that does not cause unpredictable pauses of the application needs to be employed.

The most important improvements of the RTSJ affect the following seven areas:

- thread scheduling,
- memory management,
- synchronization,
- asynchronous events,
- asynchronous flow of control,
- thread termination, and
- physical memory access.

With this, the RTSJ also covers areas that are not directly related to realtime applications. However, these areas are of great importance to many embedded realtime applications such as direct access to physical memory (e.g., memory mapped I/O) or asynchronous mechanisms.

Thread Scheduling: To enable the development of realtime software in an environment

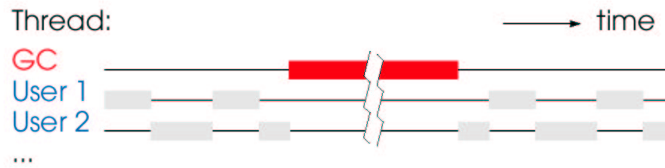

Figure 1: *In a classic Java environment, application threads are stopped by the garbage collector.*

with a garbage collector that stops the execution of application threads in an unpredictable way, new thread classes RealtimeThread and NoHeapRealtimeThread are defined. These thread types are unaffected or at least less heavily affected by garbage collection activity. Also, at least 28 new priority levels, logically higher than the priority of the garbage collector, are available for these threads. Figure 2 illustrates the new realtime thread classes that can interrupt garbage collector activity.

Memory Management: For realtime threads not to be affected by garbage collector activity, these threads need to use memory areas that are not under the control of the garbage collector. New memory classes, ImmortalMemory and ScopedMemory, provide these memory areas. One important consequence of the use of special memory areas is, of course, that the advantages of dynamic memory management is not fully available to realtime threads.

Synchronization: In realtime systems with threads of different priority levels, priority inversion situations must be avoided. Priority inversion occurs when a thread of high priority is blocked by waiting for a monitor that is owned by a thread of a lower priority. The RTSJ provides the alternatives priority inheritance and the priority ceiling protocol to avoid priority inversion.

The RTSJ offers powerful features that enable the development of realtime applications. Figure 3 shows an example how the RTSJ can be used in practice. In this example, a periodic thread is created. This thread becomes active every 20ms and creates a short output onto the standard console. A RealtimeThread is used to implement this task. The priority and the length of the period of this periodic thread need to be provided. A call to waitForNextPeriod() causes the thread to wait after the completion of one activation for the start of the next period. An introduction to the RTSJ with numerous further examples is given in the book by Peter Dibble [7].

The RTSJ provides a solution for realtime programming, but it also brings new difficulties to the developer. The most important consequence is that applications have to be split strictly into two parts: a realtime and a non-realtime part. The communication between these parts is heavily restricted: realtime threads cannot perform memory operations such as the allocation of objects on the normally heap which is under the control of the garbage collector. Synchronization between realtime and non-realtime threads is heavily restricted since it could otherwise cause realtime threads to be blocked by the garbage collector.

3 Realtime Garbage Collection

In a system that uses realtime garbage collection, this strict separation into realtime and non-realtime threads is not necessary. The strict splitting of an application is consequently not required. Threads are scheduled according to their priority or other scheduling mechanisms that are used in the system. Figure 4 illustrates such a system with the garbage collection work being performed within the application threads. There is no specific garbage collection thread that may stop the execution of other threads. We have implemented such a garbage collector as the basis of the JamaicaVM is a Java implementation that provides realtime behavior for all threads.

The realtime garbage collector performs its work predictably within the application threads. It is activated when memory is allocated. This garbage collector activity must be interruptible any time to permit preemption by more urgent threads.

The implementation of a realtime garbage collector has to solve a number of technical challenges. Garbage collector activity must be

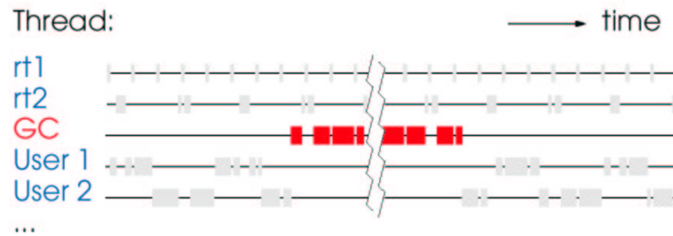

Figure 2: *RealtimeThreads in the Realtime Specification for Java are not affected by garbage collector activity.*

```
import javax.realtime.*;
public class HelloRT {        /* Periodic thread in Java: */

  public static void main(String [] args) {

    /* priority for new thread: min+10 */
    int pri = PriorityScheduler.instance().getMinPriority() + 10;
    PriorityParameters prip = new PriorityParameters(pri);

    /* period: 20ms */
    RelativeTime period = new RelativeTime(20 /* ms */, 0 /* ns */);

    /* release parameters for periodic thread: */
    PeriodicParameters perp = new PeriodicParameters(null,period,null,null,null,null);

    /* create periodic thread: */
    RealtimeThread rt= new RealtimeThread(prip, perp) {
        public void run() {
          for(int n=0; waitForNextPeriod() && (n<100), n++) {
            System.out.println("Hello "+n);
          }
        }
      };

    rt.start();    /* start periodic thread: */
  }
}
```

Figure 3: *Example programm that starts a periodic task using a RealtimeThread.*

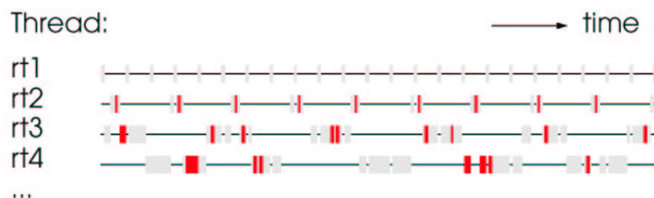

Figure 4: *Realtime Garbage Collection: All Threads are realtime threads.*

performed in very small single increments of work. In the garbage collector that is presented here, one increment of garbage collector work consists of only 32 bytes of allocated memory. On every allocation, the allocating thread has to "pay" for the allocated memory by performing a small number of these increments. The number of increments can be analyzed, such that this is possible even in realtime code.

The most difficult tasks within such an incremental garbage collector are the root scanning phase of the garbage collector, the memory fragmentation problematic and the determination of worst case execution times for successful memory allocation.

3.1 Root Scanning

Root scanning typically involves stopping of all threads such that the references stored within the runtime stacks of the threads can be found. This requires stopping of the application for times that are not acceptable in demanding realtime environments.

A means to completely avoid the root scanning phase overhead is to use a Java compiler and a virtual machine that creates additional code that ensures that all root references are copied to the heap whenever the garbage collector may become active. Specific data structures on the heap are required to hold copies of the root references. If these data structures are all made reachable from a single global root references, the root scanning phase can be reduced to scanning this single global root references. Consequently, the root scanning phase is no longer required, it is sufficient to start the garbage collector's mark phase from this single root reference. The surprising result is that the overhead for copying of the root references to the heap can be reduced by an optimizing compiler to only a few percent of the execution time [27].

3.2 Fragmentation

Once the root scanning phase has been eliminated, a simple incremental mark and sweep garbage collector can be used [8] as long as memory fragmentation issues are ignored.

However, a reliable system needs to actively fight fragmentation of memory to ensure that it can run correctly for long periods of time. Any system that allocates memory of arbitrary sizes might otherwise fail due to fragmentation of memory.

Unfortunately, defragmenting the heap through compaction of allocated memory is not easy within a realtime system. Objects typically need to be moved atomically causing long pauses since objects can be fairly large (Java objects can be arrays of arbitrary sizes).

A solution to the fragmentation problematic that completely avoids the need to compact memory is the use of fixed-size blocks that are never moved. Java objects can then be constructed out of these blocks even if they are not contiguous in memory. Figure 5 illustrates how an object can be constructed out of several blocks. Surprisingly, the use of fixed-size blocks results in performance similar to the use of a compaction mechanism since there is no need to provide for changing object locations [26].

3.3 Fixed Execution Time for GC Increments

The use of fixed-size blocks does not only provide a solution to the memory fragmentation problematic, but it also enables to limit the amount of work that needs to be done on single incremental garbage collection steps within an incremental mark and sweep garbage collector [8]. For this, the single mark and sweep steps of the garbage collector must not work on objects, but on single blocks instead.

With a block size of 32 bytes, single increments of garbage collection work then turn into marking or sweeping a single block of 32 bytes. The determination of a worst-case execution time for such a single step is straightforward [25, 28].

3.4 Configuring a Realtime Garbage Collector

To be able to determine worst-case execution times for memory allocation operations in a realtime garbage collector, one needs to know the memory required by the realtime application. With this information, a worst-case number of garbage collector increments can be determined [25]. Automatic tools can help to determine this value. The heap size can then be selected to give sufficient headroom for the

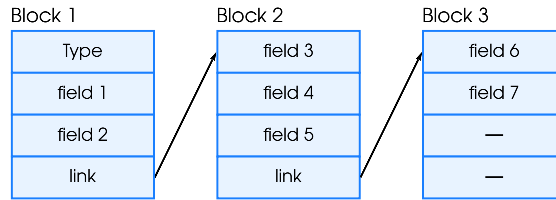

Figure 5: *Non-fragmenting object model: object with 7 fields composed out of three blocks of 16 bytes each*

garbage collector, while a larger heap size ensures a shorter execution time for allocation. Tools like the analyzer in the JamaicaVM help to configure a system and find suitable heap size and allocation times.

4 Realtime Programming

Once the unpredictability of the garbage collector has been solved, realtime programming is possible even without the need for special thread classes or the use of specific memory areas for realtime code.

4.1 Realtime Tasks

Since garbage collection activity is performed within application threads and only, when memory is allocated, a direct consequence is that any realtime task that does perform any dynamic memory allocation will not be affected by garbage collection activity at all. These realtime tasks can access objects on the normal heap exactly as all other tasks. As long as realtime tasks use a priority that is higher than other threads, they will be guaranteed to run when they are ready.

Furthermore, even realtime tasks may allocate memory dynamically. Just as any other tasks, garbage collection work needs to be performed to pay for this allocation. Since a worst-case execution time can be determined for the allocation, the worst-case execution time of the task that performs the allocation can be determined as well.

4.2 Communication

The communication mechanisms that can be used between threads with different priority levels and timing requirements are basically the same mechanisms as those used for normal Java threads: shared memory and Java monitors.

4.2.1 Shared Memory

Since all threads can access the normal, garbage collected heap without suffering from unpredictable pauses due to garbage collector activity, this normal heap can be used for shared memory communication between all threads. Any high priority task can access objects on the heap even while a lower priority thread accesses the same objects or even while a lower priority thread allocates memory and performs garbage collection work. In the latter case, the small worst-case execution time of an increment of garbage collection work ensures a bounded and small thread preemption time, typically in the order of a few microseconds.

4.2.2 Synchronization

The use of Java monitors in *synchronized* methods and explicit *synchronized* statements enables atomic accesses to data structures. These mechanisms can just as well be used to protect accesses that are performed in high priority realtime tasks and normal non-realtime tasks.

However, the standard Java semantics for monitors do not prevent from priority inversion that may result from a high priority task trying to enter a monitor that is held by another task of lower priority. The stricter monitor semantics of the Realtime Specification for Java [5] avoid this priority inversion. All monitors are required to use priority inheritance or the priority ceiling protocol, such that no priority

inversion can occur when a thread tries to enter a monitor.

As in any realtime system, the develop has to ensure that the time that a monitor is held by any thread must be bounded if this monitor needs to be entered by a realtime tasks that requires an upper bound for the time required to obtain this monitor.

4.3 Standard Data Structures

The strict separation of an application into a realtime and non-realtime tasks that is required when the Realtime Specification for Java is used in conjunction with a non-realtime garbage collector makes it very difficult to have global data structures that are shared between several tasks.

The Realtime Specification for Java even provides special data structures such as *WaitFreeWriteQueue* that enable communication between tasks without the need to synchronize and running the risk of introducing priority inversion.

In a system that uses realtime garbage collection, such specific structures are not required. High priority tasks can share standard data structures such as *java.util.Vector* with low priority threads.

5 Related Work

A significant amount of work was carried out in the area of automatic memory management. This section can only give a short overview of the publications relevant to real-time systems.

5.1 Overview on Garbage Collection Techniques

For a general overview on garbage collection techniques, the book written by Jones and Lins is a good start [20]. Wilson compiled a comprehensive survey of garbage collection techniques for single processor machines [32].

Bengtsson and Magnusson give a short overview of realtime garbage collection techniques (copying, mark-and-sweep, generational), comparing the efficiency for different heap configurations [3]. A good survey of non-garbage collected dynamic storage allocation techniques is given by Wilson et. al. [34].

5.2 Incremental and Concurrent Mark-and-Sweep Garbage Collectors

An emphasis of previous research was to enable that a garbage collector runs concurrently with the main application program (the mutator) or performs its work in small increments. Widely applied is the three-color marking described by Dijkstra et. al. [8]. The algorithm uses two phases: mark and sweep. The first phase marks all reachable memory, while the second phase adds the remaining unmarked memory to the free list.

Queinnec et. al. suggest running the mark and sweep phases concurrently (mark-during-sweep), so that two threads or processors are performing garbage collection and the overall performance of the system is improved [23]. Wallace and Runciman [30] combine this approach with a small stack for marked objects in their incremental collector that is supposed to be used in embedded real-time systems. Unfortunately, the quadratic worst-case execution time of their implementation disqualifies it for many applications. Huelsbergen and Winterbottom further improve mark-during-sweep by avoiding fine-grain synchronization [18].

A promising extension of Dijkstra's mark-and-sweep garbage collection algorithm for multiprocessor systems is presented by Doligez and Gonthier [9]. Unlike Dijkstra's algorithm, this algorithm permits several concurrent application threads. A more complex write barrier and a sophisticated synchronization mechanism at the beginning of a GC cycle enable parallel garbage collection without fine-grain synchronization.

Domani et. al. have extended the algorithm by Doligez and Gonthier with Java-specific features and implemented it for Java [11, 12]. The algorithm was extended to support finalization, weak references and (String-) intern tables as required by Java.

Dubé, Feeley and Serrano present a realtime garbage collector [13]. The approach uses a mark-compact scheme that incrementally moves objects. The algorithm was implemented within a rudimentary Scheme interpreter. Some basic runtime measurements were made using a Fibonacci function written in Scheme and comparing the results to

a non-real-time Cheney-style garbage collector [6]. The real-time garbage collector causes a slowdown of a factor 3 in this simple test.

5.3 Two-Space Copying Garbage Collectors

Baker's copying real-time collector [2] is the first GC to provide real-time guarantees on allocation (CONS operations in LISP) coupling garbage collection work and allocation and using knowledge of the memory requirements of the system to control the amount of garbage collection work. The heap is divided into two semi-spaces: from-space and to-space. During collection, all live objects from from-space are evacuated into to-space. When all objects reside in to-space, the role of both spaces is swapped (the 'flip') and the next collection cycle begins. Drawbacks of the approach are high memory demand for the two semi-spaces (from-space and to-space) and bad runtime overhead due to the use of read barriers. Baker also proposes generalizing the mechanism for objects of arbitrary sizes (such as arrays) by incrementally copying them.

Wilson and Johnstone [33] have further improved Baker's algorithm avoiding the overhead caused by a read barrier used in the original approach and using segregated storage to manage different sized chunks of memory. They provide no good solution for fragmentation in their approach either. Limiting the allocation size to powers of two reduces the worst case loss due to fragmentation at a high cost of internal fragmentation (padding of chunks to the next power of two), while worst-case fragmentation can still be large.

5.3.1 Generational and Age Based Collectors

Many so-called generational garbage collector implementations are based on the heuristic that old objects tend to live longer than younger ones. This idea was first presented by Lieberman and Hewitt [22]: Generational collectors restrict frequent collections to a subset of the total heap that contains the youngest objects (the youngest generation) to reduce the average pause times and total garbage collection overhead. A major problem of this approach in a system that requires real-time guarantees is that it is based on heuristics that may not hold for the actual application. The original generational heuristic together with the experience that references tend to link young objects with old objects more often than old objects with young objects do not hold for all applications. For the application of generational garbage collection in a hard real-time system, a quantitative measure needs to be found to determine the degree to which a given application behaves as expected by the heuristics.

A 'real-time' implementation of a generational collector was presented by Engelstad and Vandendorpe [14]. Collection work is coupled with allocation and based on heuristics that control the amount of collection work so that the system is less likely to run out of memory. However, the scheme doesn't seem to give any guarantee that enough collection work will be performed for the system not to run out of memory. The authors have measured that this implementation bounds garbage collection pause times to 0.5-10ms, which is just enough to use it for interaction with humans, but not for most technical applications of real-time controllers. Even if one considers the higher processor speeds that are available today, one can expect that this approach will not perform significantly better on todays hardware taking the larger memory sizes into account that are used in today's systems.

A 'quasi real-time' generational collector was presented in [10] for Concurrent Caml Light, a multi-threaded implementation of ML. A crucial prerequisite for this implementation is the notion of mutable and immutable objects in ML. The latter can be duplicated, so that each mutator (thread) can have its own private young generation heap that holds copies of these objects. Other objects are allocated in a shared heap, the old generation. Each thread performs the garbage collection of its own private young generation. A separate thread is responsible for collecting the shared heap using an extension of the Dijkstra et. al. algorithm [8]. This approach ensures an upper bound for the pause times of private collections, but it cannot guarantee sufficient progress of the collection of the shared heap, requiring unpredictable stopping of threads to wait for the termination of this collection or extension of the address space whenever the garbage collector does not catch up with the allocation speed of the application.

Recent research indicates that the heuris-

tic that generational collection is based upon may be improved [29]. The basic idea of this age-based garbage collection is first to sort objects by their age measured in allocation time. Then the collection should be limited to the subrange of the allocated memory that has the highest object 'mortality'.

5.4 Realtime Reference Counting

Ritzau describes a variant of reference counting that can be employed for realtime systems [24]. His scheme combines lazy reference counting [31] with the use of fixed-size blocks [26] in a way that results in constant time memory allocation. As all reference counting schemes, this approach suffers from the inability to reclaim cyclic data structures.

Blackburn and McKinley [4] combine reference counting with copying generational garbage collector to achieve short pause times and high garbage collector throughput.

5.5 Scheduling Garbage Collection

Some research has also been done in the area of scheduling garbage collection in a real-time system: In the system proposed by Roger Henriksson [15, 16], real-time guarantees for memory management of some high-priority processes can be given as long as low-priority processes that are interleaved with the garbage collection process obtain sufficient CPU time. This time has to be sufficient so that the system can perform enough recycling work to satisfy the allocations performed in the high-priority processes. Henriksson bases his collector on Baker's algorithm [2] and reserves enough space in the target to-space so that high-priority processes can allocate their objects even right before a flip of from- and to-space. Kim et. al. [21] describe a way to schedule a garbage collection task in a system with periodic and sporadic real-time tasks using the sporadic server approach. The scheduling requires knowledge on the allocation behavior (amount of allocation within one period) of all real-time tasks to ensure schedulability of the garbage collector.

6 Conclusion

The extensions defined by the Realtime Specification for Java enable the use of Java in new application domains that have strict requirements on the timing behavior of the application.

However, the need to use special thread classes and specific memory areas that are not under the control of the garbage collector and not accessible by normal threads complicates the development of larger systems that require communication and shared data between realtime and non-realtime code.

The use or realtime garbage collection together with the extensions defined in the Realtime Specification for Java makes it possible to overcome these restrictions and provides a more straightforward and simpler development of realtime code using Java. Even systems that do not require dynamic memory management within realtime tasks can profit from such a system. The development of realtime code becomes simpler, such that higher productivity and higher software quality can be expected. Such a system provides the advantages, that made Java so successful, to the developer of realtime systems.

References

[1] Aero-vm, the hard realtime virtual machine for onboard space systems. www.aero-vm.com, 2003.

[2] Henry G. Baker. List processing in realtime on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[3] Mats Bengtsson and Boris Magnusson. Real-time compacting garbage collection. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[4] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior referene counting: Fast garbage collection without a long wait. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN

Notices, Anaheim, CA, November 2003. ACM Press.

[5] Greg Bollela. *Real-Time Specification for Java*. Addison-Wesley, 2001.

[6] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[7] Peter C. Dibble. *Real-Time Java Platform Programming*. Prentice Hall, 2002.

[8] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[9] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[10] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[11] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[12] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. Technical Report 88.385, IBM Haifa Research Laboratory, 2000. Fuller version of [11].

[13] Danny Dubé, Marc Feeley, and Manuel Serrano. Un GC temps réel semi-compactant. *Journées Francophones des Langages Applicatifs*, pages 165–181, January 1996.

[14] Steven L. Engelstad and James E. Vandendorpe. Automatic storage management for systems with real time constraints. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[15] Roger Henriksson. Predictable automatic memory management for embedded systems. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[16] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[17] Hidoors, high integrity object-oriented realtime systems. www.hidoors.org, 2002-2004.

[18] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In Jones [19], pages 166–175.

[19] Richard Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.

[20] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[21] Taehyoun Kim, Naehyuck Chang, Namyun Kim, and Heonshik Shin. Scheduling garbage collector for embedded real-time systems. In *ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, pages 55–64, Atlanta, GA, May 1999. ACM Press.

[22] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[23] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING Sweep rather than Mark THEN Sweep. *Lecture Notes in Computer Science*, 365:224–237, 1989.

[24] Tobias Ritzau. *Memory Efficient Hard Real-Time Garbage Collection.* PhD thesis, Linköping University, May 2003.

[25] Fridtjof Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In Jones [19], pages 130–137.

[26] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, November 2000.

[27] Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *Tenth International Conference on Compiler Construction (CC2001)*, Genoa, April 2001.

[28] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages.* aicas Books, 2002.

[29] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms.* PhD thesis, University of Massachusetts, 1999.

[30] Malcolm Wallace and Colin Runciman. An incremental garbage collector for embedded real-time systems. In *Proceedings of the Chalmers Winter Meeting*, pages 273–288, Tanum Strand, Sweden, 1993. Published as Programming Methodology Group, Chalmers University of Technology, Technical Report 73.

[31] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[32] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[33] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[34] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.