Memory Management for Real-time Java: State of the Art

Filip Pizlo Jan Vitek
Purdue University

Abstract

The Real-time Specification for Java extends the Java platform to support real-time processing and introduces a region-based memory model, called scoped memory, which side-steps the Java garbage collector. While scoped memory succeeds in protecting real-time tasks from execution time jitter, practical experience points to shortcomings. This paper takes stock of the state of the art in memory management for RTSJ programs.

1. Introduction

Managing memory is a crucial part of most software systems. Real-time systems have traditionally relied on manual memory management in programming languages such as C which provide unfettered access to memory. By contrast, modern object-oriented languages take the control of memory away from programmers and guarantee memory safety through garbage collection. While there are undeniable software engineering benefits to memory safety in terms of productivity, reliability and security, performance is perceived to be inferior with pauses that are not acceptable in hard-real-time applications.

The Real-time Specification for Java (RTSJ) [16] extends the Java programming language with the functionality needed for real-time processing. The RTSJ designers recognized that without a memory management solution that would both preserve memory safety and ensure reliable, jitter-free execution, the whole endeavor was destined for failure. At the time, real-time garbage collection was still in its infancy; some algorithms had been published [36, 37, 40, 52] but they were, by and large, still untested. Instead the RTSJ designers decided to offer a region-based memory model [60]. Memory is arranged as a tree of regions, called *scoped memory areas* [17]. Objects needed by real-time tasks can be safely allocated in regions where they will be protected from garbage collector interference and then deallocated in bulk. In a departure from earlier work, safety is not enforced by the compiler; run-time checks are used instead.

With the development of open source [51] as well as commercial real-time Java virtual machines [15, 4, 1], the RTSJ has become a viable alternative for large real-time systems. Boeing and Purdue successfully deployed a RTSJ controlled UAV [3]. IBM and Raytheon are using Java to develop a comprehensive battleship computing environment [33]. Other notable applications are in audio processing [5, 35], industrial control [28], trading software, and visualization. Experience implementing [10, 23, 44, 3] and using [14, 41, 12, 46, 50] scoped memory revealed a number of drawbacks. First, there are noticeable overheads to performing the mandated run-time checks. Second, space usage is not optimal as objects cannot be freed individually. And finally, programming with scoped memory is error prone, leading to increased development effort. This has spurred interest in alternatives that can complement scoped memory. For instance, real-time garbage collection has made substantial advances [8, 4, 55, 30, 25]. There has also been work on increasing robustness of scoped memory programs by means of a statically verifiable type discipline [18, 63, 2]. A number of alternative programming models tailored to particular classes of applications are also being investigated [56, 57, 6, 58].

2. The cost of memory management

Any choice of memory management discipline involves a tradeoff between programmer productivity, effective use of resources, and predictability.

2.1. Programmer productivity

Manual memory management requires more development effort than garbage collection due to the need to deal with deallocation, as well as memory access errors that could lead to unbounded heap corruption. Scoped memory is somewhat better than manual memory management—heap corruption is not possible, but memory access exceptions may be thrown when “unsafe” references are about to be created. Scoped memory forces a style of programming where objects are not freed individually. This is a double-edged sword: while it often means writing less code to free objects, it also forces the programmer to think carefully about which scope to use based on the intended lifetime,

and the intended reference pattern, of an object. This entails additional development effort; existing libraries must often be restructured to include scoped memory awareness.

2.2. Performance

Common wisdom holds that manual memory management is faster than garbage collection and also that custom allocators are more efficient then general purpose ones [38, 39]. The reality is more subtle. To evaluate the impact of different memory management regimes, one must take data locality and cache behavior into account. A collector can, for instance, improve locality by compacting the data. Many collectors allow contiguous allocation of data which is fast and has good locality. A generational collector can outperform (by up to 9%) manual memory management when space is plentiful [31]. But if the available space decreases, the performance of garbage collected programs decreases by up to 70% over non-GCed programs. Most custom allocators do not improve performance; the exception is region-based allocation, which can yield a 44% speed-up over a general purpose allocator [13]. In terms of space usage, region based allocation was found to increase memory consumption by up to 230% due to the inability to free individual objects. A GCed system typically requires between 2 to 5 times more memory than manual memory management [31]. However, for long running applications, such as most real-time systems, GCs have the advantage of being able to compact memory and thus avoid pathological cases of fragmentation.

2.3. Predictability

The goal of any real-time system is to ensure that all tasks meet their deadlines. This requires understanding and bounding memory management jitter. In the absence of GC, programmers manage memory via a simple interface that consists of allocation and deallocation operations. These operations can easily be bounded in time (for example, with a buddy allocator). Space bounds are more tricky due to fragmentation; the accepted techniques tend to involve either the prevention of fragmentation by static allocation or overprovisioning.

In most garbage collectors, allocations can in the worst case result in a full traversal of the entire heap. This presents a problem: if the collector is given the freedom to interrupt or prolong the execution of heap operations, how can we reason about worst-case behavior? A number of collectors attempt to solve this problem—ranging from tight bounds on the length of interruptions of each memory operation [42], to attempts to hide all collector activity inside the slack in the real-time schedule [30]. More recent approaches [8, 4, 49] rely on a particular understanding of RTGC, in which collector interruptions are viewed as benign if they occur with well-known periodicity. There is no agreement as to which approach is best. Worse, there is no agreement on *what it means for a collector to be real-time*.

Collector Pause Time. When measuring collector responsiveness, the most commonly advertised metric is pause time. This metric makes sense for collectors that perform all of their work at once. But most real-time collectors split work between multiple pauses, meaning that we are more interested in the total effect incurred by all of the pauses within a specific release of a task. The pause-time obsession has driven some GC designers to strategies in which there are no pauses [30, 47]. These collectors still retain allocation procedures that take non-zero time, and have read and/or write barriers that can prolong the execution of tasks by a non-trivial, *and often unreported*, amount.

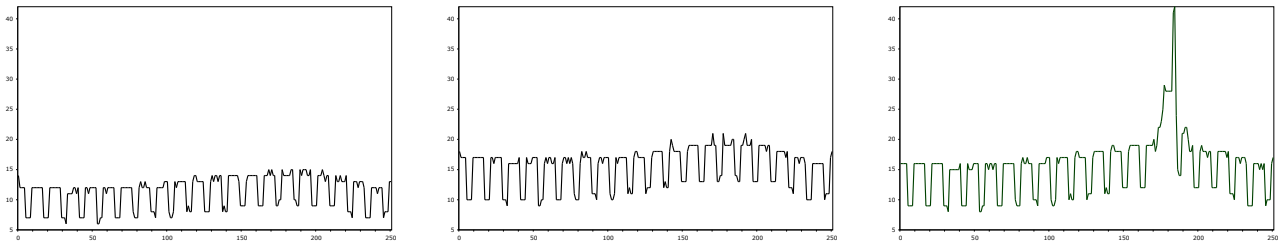
Minimum Mutator Utilization. Cheng and Blelloch developed a metric for predictability—the *minimum mutator utilization* or MMU [20]. MMU is represented as a graph in which the x-axis is the time spent executing a task, and the y-axis is the minimum fraction of that time spent doing useful application work (or "mutator" work). A MMU of 50% means that a task will spend, at most, half of its time doing garbage collection. But is MMU useful for comparing different real-time collectors? Collectors often mandate special compilation strategies that introduce non-trivial amounts of overhead. Typically, these overheads cannot be treated as *pauses* or *interruptions*—they are simply too fine-grained. Thus, MMU quantifies only a portion of collector overheads. Figure 1 illustrates this: running the program without GC is faster across the board than running with the RTGC at 50% MMU. Further, the worst-case under RTGC is 2.8 times greater than the worst-case without GC, leading to an "effective" MMU of 38%. (Note that the target MMU depends on the application, it has been reported that a MMU target of 75% suffices for many real applications.)

Worst Case Execution Time. Perhaps the most sensible approach to understanding RTGC is to consider the worst-case execution time. This is not a new idea. Nilsen [42] showed that his collector has $O(1)$ heap accesses and $O(n)$ allocations. If an analytical analysis of the worst-case time spent in reads, writes, and allocations is possible, it is likely "as good as it gets" when it comes to real-time performance metrics. MMU is then only useful if the collector incurs costs outside of heap operations.

3. Region-based allocation

3.1. Scoped Memory

In the RTSJ, storage for an allocation request performed by a real-time task is serviced differently from standard Java allocation. The RTSJ extends the Java memory management model to include dynamically checked regions known as *scoped memory areas* represented by



(a) Region-based allocator, no run-time checks.

(b) Region-based allocator, RTSJ run-time checks.

(c) RTGC, 50% MMU, 1ms max pause times.

Figure 1: Performance of a 10Hz periodic real-time task [50]. The x-axis shows releases, the y-axis time to complete a release. The worst observed case for were 15ms (a), 21ms (b), and 41ms (c). The difference between (a) and (b) is due to the presence of RTSJ read and write barriers. The average times were 11.2ms (a) and 15.4ms (b and c). Throughput is no worse with GC than with scoped memory on an Athlon, 1.6GHz, 1GB, Linux 2.4.7-timesys-3.1.214, with Ovm and GCC 4.0.1.

subclasses of `ScopedMemory`. A memory area is an allocation context that provides a pool of memory shared by threads executing within it. Individual objects allocated in a memory area cannot be deallocated; instead, an entire area is torn down as soon as all threads exit it. The RTSJ defines two distinguished scopes: *immortal* and *heap* memory, respectively for objects with unbounded lifetimes and objects that must be garbage collected. Two new kinds of threads are also introduced: *real-time* threads that are scheduled by a real-time scheduler and may access scoped memory areas (`RealtimeThread`); and *no heap real-time* threads (`NoHeapRealtimeThread`), which are real-time threads protected from pauses. Dynamically enforced safety rules check that a memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope with a shorter lifetime. This means that heap memory and immortal memory cannot hold references to objects allocated in scoped memory, nor can a scoped memory area hold a reference to an object allocated in an inner scope. Memory areas have an `enter()` method that permits application code to execute within a scope. Using nested calls, a thread may enter multiple scopes, dynamically building up a scope hierarchy. Misuse of these methods is punished by dynamic errors; `IllegalAssignmentError` and `MemoryAccessError` are thrown on attempted violations of the memory access rules. Reference counting on `enter`s ensures that objects allocated in a scope are reclaimed and finalized when the last thread leaves the scope.

The cost of using scoped memory in Ovm [3] is predictable. Read and write are $O(1)$. Write barriers are added to every assignment into a field of reference type (i.e. `Object` or one of its subtypes). There is a fast path, inlined by the compiler, for the case of local writes and slow path that uses range checks [44]. Constant-time read barriers are added to loads [50]. Objects are allocated out of contiguous blocks of memory. The cost of `new` is $O(n)$ due to memory zeroing. No additional space is required to store scope pointers, as the VM instead maps pages to scopes. Other design choices are discussed in [23, 24, 10].

As there is no isolation between real-time and non-real-time parts of the system, if a real-time task tries to acquire a monitor held by a non-real-time thread, the real-time task may experience unbounded locking in case the Java thread triggers GC. A less catastrophic source of blocking is due to finalization. In a scope, finalization occurs when the last thread exits and no thread may re-enter the scope until finalization is complete. With GC, the problem is less severe as finalization can be offloaded to a separate thread.

Programming with scoped memory entails a loss of compositionality. Components, when tested independently, may work just fine, and break when put in a particular scoped memory context. This is because scoped memory adds an extra dimension—*where* each object has been allocated—that complicates reasoning about program correctness. Design patterns and idioms for programming effectively with scoped memory are discussed in [46, 11, 14]. Following these guidelines simplifies RTSJ development.

3.2. Scoped Types

While the RTSJ designers chose to enforce memory safety with run-time checks, this is not the only alternative. Memory-safe region-based allocation does not require run-time checks [27, 61, 29]; instead, regions can be added to the language's type system. The first region types to support subtyping and inheritance appeared in [21]. A more recent strand of work seeks to adapt the notion of ownership types of [43] to deal with scoped memory [18]. Scoped types [63, 2], for instance, are an ownership-based approach which does not require changes to the source language.

3.3. Safety Critical Java

The on-going safety critical Java standardization effort [34] seeks to define a subset of the JSR that will be easier to validate and prove correct in the context of safety-critical applications. Correctness of memory operations has been identified as one of the key concerns. To address this, the expert group is considering a variant of scoped types [2] that is backwards compatible with the RTSJ and VM-independent.

3.4. Alternative Programming Models

Some of the complexities and pitfalls of the RTSJ can be attributed to the desire of a general purpose programming model for real-time systems. A number of recent results offered programming models that geared to particular tasks. Eventrons [56] are a programming abstraction that target highly responsive systems. They require very little support from the virtual machine, provide static safety guarantees and have been shown to be adequate for running tasks at periods as low as 45μsecs. An Eventron is a group of objects allocated in non-GCed memory with an associated real-time task. Safety is guaranteed by an analysis that runs at Eventron instantiation. Reflexes [57] relax some of the constraints of Eventron: they allow allocation in a special region that is cleared after each release and, instead of initialization-time analysis they are validated by a type checker similar to [2]. Exotasks [6] and StreamFlexes [58] extend the Eventron/Reflex programming model to graphs of tasks connected by data channels.

4. Real-time Garbage collection

Garbage collection removes some of the burden of memory management from the application programmer by automatically freeing all or most of the memory that is no longer in use. But high performance collectors are typically designed to free all unused objects in one step. Such long "pauses" lead to unpredictable timing behavior, with tasks occasionally taking much longer than expected. Further, much of GC research has been concerned with optimizing for throughput rather than uniform overhead and responsiveness. However, recent work on RTGC has lead to systems that exhibit levels of predictability suitable for many real-time applications. Some of these collectors have been productized [53, 4, 59].

4.1. Application-Collector Interaction

RTGCs split collector work into small increments. This implies allowing the application to run even as the collector is in the middle of traversing the heap in search of objects that are "reachable," that is, objects that should not be freed. But allowing the application to run while graph reachability is being determined is hard—the application may arbitrarily change the shape of the graph, in the worst case leading to the collector freeing objects prematurely. More difficulties are added if the collector has the power to defragment the heap. Defragmentation involves moving objects—thus in an incremental defragmenting collector, the application must be able access objects while they are in motion. Both of these issues—the application changing the shape of the graph and the collector moving objects used by the application—are typically solved with compiler-inserted *barriers*. A barrier is a hint to the compiler/runtime given that a heap access operation should be compiled specially. Typical examples include the Brooks barrier [19], where every heap access performs an additional indirection; these barriers are often sufficient for allowing the application to use objects even as they are being copied. More sophisticated barriers include the Yuasa barrier [62], which replaces writes of object reference fields in the heap with a short code snippet that is sufficient for keeping the collector up-to-date on heap changes, thus preventing any reachable objects from being freed.

Production RTGCs such as Bacon et al's Metronome [4] typically use variants of the Brooks and Yuasa barriers, as this is the fastest known combination for ensuring sound application-collector interaction. However, some collectors, such as the one originally proposed by [30] are known to use much heavier barriers, where reads of object reference fields in the heap are replaced with longer sequences of code. Also, [55] uses barriers on assignment of pointer local variables, which is likely to cause a significant throughput hit but is very effective in reducing worst-case pause times due to stack scanning [54]. See Table 1 for the code typically used in the Metronome, Henriksson, and JamaicaVM barriers. For a thorough review of barrier techniques, see [45].

4.2. Collector Scheduling

The main property that differentiates real-time collectors from traditional collectors is scheduling. Whereas a traditional collector preempts the mutator once, a RTGC will split up its work into increments allowing the mutator to make progress even if the collector is not finished. But how are these collector increments scheduled? A poor choice of scheduling strategy can be devastating, leading to too much overhead or preventing the collector from freeing memory quickly enough to keep up with the mutator's demands. Early incremental collectors such as [9] taxed all heap operations—including, e.g., reads—with increments of collector work. In the worst case, the execution time of every memory read in a real-time task would include the job of copying an entire object. This is unacceptable in modern systems, as heap reads are frequent and objects may be large. Thus, modern RTGCs have more clever scheduling strategies, which we review next.

Fine-grained Work-based. A *work-based* collector taxes units of application work with units of collector work. The Baker collector [9], for example, will make a small unit of progress on every heap read and a larger unit of progress on

Operation / Barrier	No Barriers	Metronome	Henriksson	JamaicaVM
Local Reference Assignment	`a = b`	`a = b`	`a = b`	`mark(b)` `a = b`
Primitive Heap Load	`a = b->f`	`a = b->forward->f`	`a = b->forward->f`	`a = b->f`
Reference Heap Load	`a = b->f`	`a = b->forward->f`	`a = b->forward->f` `mark(a)`	`a = b->f` `mark(a)`
Primitive Heap Store	`a->f = b`	`a->forward->f = b`	`a->forward->f = b`	`a->f = b`
Reference Heap Store	`a->f = b`	`mark(a->forward->f)` `a->forward->f = b->forward`	`mark(b)` `a->forward->f = b->forward`	`mark(b)` `a->f = b`

Table 1: *Code for different barrier types. The Metronome [8, 4] uses a variation of the Yuasa barrier in combination with a Brooks barrier. The Henriksson [30] collector uses a Brooks barrier in combination with barriers on reference heap loads and stores that accomplish the same goals as the Yuasa barrier but at a higher overhead. The JamaicaVM [55] barriers are the heaviest (requiring local variable assignment to be instrumented), but can be used to aggressively reduce pause times.*

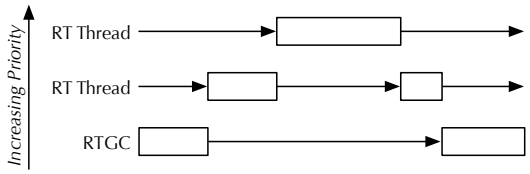


Figure 2: *Slack-based RTGC scheduling. The RTGC only runs when none of the real-time threads are running.*

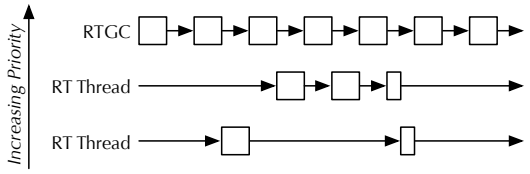


Figure 3: *Time-based RTGC scheduling. The RTGC has the highest priority but only interrupts the application at well-defined intervals.*

every heap allocation. But if the goal is to ensure that the collector "keeps up" with allocation demand, then taxing allocation requests is sufficient. This is the strategy taken by modern real-time work-based collectors [55]. We refer to this scheme as *fine-grained work-based* because it does not allow for amortization—e.g., imposing overheads on groups of allocation requests rather than individual requests. With a fine-grained collector interference is highly predictable. But, the lack of amortization, and the need for very small increments results in a constrained collector design. For example, JamaicaVM splits objects and arrays into 32-byte blocks causing space overheads due to round-up and time overheads due to additional indirections.

Slack-based. With Henriksson's *slack-based* [30] approach, the collector works only when real-time tasks are not running. This allows real-time tasks to run fast, yielding the smallest pause times and the highest MMU, but requires that the programmer provisions for the collector when scheduling the system. Slack-based collectors require an accurate characterization of the allocation rate of the application. As slack is typically longer than the short increments of a fine-grained work-based collector, algorithms are somewhat less constrained—the collector may assume that a larger amount of collector activity is likely to proceed without interruption. A variant of Henriksson's approach is used in [59].

Time-based. Instead of taxing application work or requiring that slack be left in the schedule for the collector to run, Bacon et al's *time-based* [8, 4] approach schedules the collector at regular predetermined intervals. The collector is configured *a priori* with two tuning parameters: the length of the collector interval c and the length of the application (mutator) interval m. The collector always allows the application to run for at least m time without interruption, and never interrupts the application for longer than c. This yields predictable pause times and an MMU that converges to $m/(c+m)$. Time-based collectors require a characterization of the worst-case allocation rate. Leaving slack for the collector is not required; instead, the collector will steal time from the application according to the preset schedule. Unlike both work-based and slack-based designs, time-based systems have a well-known and typically long collector increment size (ranging from $12ms$ [8] to under a millisecond [4, 49]), which tends to make the collector easier to design.

Amortized work-based. Time-based and slack-based approaches both tend to have longer collector increments than the fine-grained work-based approach, leading to a more efficient collector. But the work-based approach is attractive as it always keeps up with any allocation rate. This begs the question—what if we allowed amortization? For example, every N bytes allocated we may perform $O(N)$ collector work. Thus, allocation would still be linear in time under an amortized worst-case analysis. Choosing a large N leads to long collector increments and a simpler design, while choosing a small N reduces pause times while constraining the collector. The most extreme example of amortized work-based collectors is the traditional stop-the-world col-

lector, where $N = M_T - M_L$, where M_T is the total size of the heap and M_L is the amount of bytes that the application is using. While this may appear undesirable for real-time systems, even the stop-the-world design has found its uses in applications that have a high demand for predictability: the JAviator [6] uses heaps managed by a non-incremental collector.

Concurrent. Concurrent collectors do all of their work on a separate set of processors. The resulting design promises little or no collector interference, but like time-based and slack-based systems requires knowing the allocation rate. Concurrent RTGCs are likely to become more common as multiprocessing becomes more prevalent.

4.3. New Developments

RTGCs still incur high overheads relative to both manual memory management and non-real-time garbage collectors due to a combination of expensive barriers, memory overheads, and the need for more complex collection algorithms. But overall overheads can be reduced by reducing the amount of collector work. Examples of collectors that aim to reduce collector work are the Syncopated Metronome [7], the Generational Metronome [26], and the Hierarchical Real-time Garbage Collector [49]. The latter collects different parts of the heap (heaplets) at different priorities. For example, a private heaplet may be given to real-time tasks, with those tasks only yielding to the private heaplet's collector. As this heaplet will be smaller than the total heap, which may include non-real-time state, and is likely to have a lower allocation rate (since non-real-time allocation is not included), the length and frequency of collector interruptions experienced by the real-time tasks is reduced.

Combining multiprocessing and RTGC continues to be an exciting area of research. Ideally, adding an RTGC to a multiprocessor application should not increase the amount of inter-processor communication, nor should increasing the number of processors have an adverse effect on pause time. Current RTGCs are far from this ideal. For example, the Metronome can be extended to a multiprocessor setting by having all processors yield to the collector during a collector increment. But as the number of processors grows, so too the amount of work required to stop all threads grows. However, there is a growing body of work that aims to produce a multiprocessor RTGC in which it is not necessary to stop all threads. The first collector to claim support for multiprocessor real-time is [20]. This work targets the ML programming language and exploits the property that heap objects are rarely modified. It requires the use of locking to synchronize between the collector and application, which leads to a non-trivial amount of worst-case overhead. Sapphire [32] is the first Java collector with high performance on multiprocessors, but relies heavily on the Java memory model and does not provide strong timing guarantees for applications that use non-blocking algorithms. Stopless of Pizlo et al [47] is the first collector to support strong time bounds even if non-blocking algorithms are used. Pizlo et al further show how a probabilistic understanding of time bounds can be used to enhance RTGC performance on multiprocessors [48]. Click et al. [22] show how custom hardware can be used to create a soft real-time collector.

5. Conclusion

At present, doing memory management in real-time Java places the burden on the programmer to choose one the of many programming models which we have reviewed. A careful choice is likely to lead to excellent software engineering benefits, and in some cases, performance that is on-par with C code [56]. But as we have shown, research into RTGCs is coming increasingly close to the goal of a universal programming model—where safety and easy development carry no runtime cost.

References

[1] AICAS, *The Jamaica virtual machine,* www.aicas.com.

[2] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao, *Scoped types and aspects for real-time Java memory management*, Realtime Systems Journal, 37, 2007.

[3] A. Armbuster, J. Baker, A. Cunei, D. Holmes, C. Flack, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, *A Real-time Java virtual machine with applications in avionics*, ACM Transactions in Embedded Computing Systems, 2007.

[4] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley, *Design and implementation of a comprehensive real-time Java virtual machine*, in Conference on Embedded software (EMSOFT), 2007.

[5] J. Auerbach, D. F. Bacon, F. Bömers, and P. Cheng, *Real-time music synthesis in Java using the Metronome garbage collector*, in International Computer Music Conference, 2007.

[6] J. S. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer, *Java takes flight: time-portable real-time programming with exotasks*, in Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2007.

[7] D. F. Bacon, P. Cheng, D. Grove, and M. T. Vechev, *Syncopation: generational real-time garbage collection in the metronome*, in Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2005.

[8] D. F. Bacon, P. Cheng, and V. T. Rajan, *A real-time garbage collecor with low overhead and consistent utilization*, in Symposium on Principles of Programming Languages (POPL), 2003.

[9] H. G. Baker, *List processing in real time on a serial computer*, Communications of the ACM, 21, 1978.

[10] W. S. Beebee, Jr. and M. Rinard, *An implementation of scoped memory for Real-Time Java*, in Embedded Software Implementation Tools for Fully Programmable Application Specific Systems (EMSOFT), 2001.

[11] E. G. Benowitz and A. Niessner, *A patterns catalog for RTSJ software designs*, in Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), 2003.

[12] E. G. Benowitz and A. F. Niessner, *Experiences in adopting real-time java for flight-like software*, in International workshop on Java technologies for real-time and embedded systems (JTRES), 2003.

[13] E. Berger, B. Zorn, and K. McKinley, *Reconsidering custom memory allocation*, in Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002.

[14] G. Bollella, T. Canham, V. Carson, V. Champlin, D. Dvorak, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz, *Programming with non-heap memory in the real-time specification for Java*, in Companion of the Conference on Object-oriented programming, systems, languages, and applications, 2003.

[15] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain, *Mackinac: Making hotspot real-time*, in International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 2005.

[16] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[17] G. Bollella and K. Reinholtz, *Scoped memory*, in International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 2002.

[18] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard, *Ownership types for safe region-based memory management in real-time Java*, in Conference on Programming Language Design and Implementation (PLDI), 2003.

[19] R. A. Brooks, *Trading data space for reduced time and code space in real-time garbage collection on stock hardware*, in Proceedings of the ACM Conference on Lisp and Functional Programming, 1984.

[20] P. Cheng and G. E. Blelloch, *A parallel, real-time garbage collector*, in Conference on Programming language design and implementation (PLDI), 2001.

[21] M. V. Christiansen, F. Henglein, H. Niss, and P. Velschow, *Safe region-based memory management for objects*, tech. rep., DIKU, University of Copenhagen, 1998.

[22] C. Click, G. Tene, and M. Wolf, *The pauseless gc algorithm*, in International Conference on Virtual execution environments, 2005.

[23] A. Corsaro and R. Cytron, *Efficient memory reference checks for real-time Java*, in Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES), 2003.

[24] A. Corsaro and D. Schmidt, *The design and performace of the jRate Real-Time Java implementation*, in International Symposium on Distributed Objects and Applications (DOA), 2002.

[25] D. Detlefs, *A hard look at hard real-time garbage collection.*, in International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 2004.

[26] D. Frampton, D. F. Bacon, P. Cheng, and D. Grove, *Generational real-time garbage collection*, 2007.

[27] D. Gay and A. Aiken, *Language support for regions*, in Conference on Programming Language Design and Implementation (PLDI), 2001.

[28] S. Gestegard Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, *Using real-time Java for industrial robot control*, in International Workshop on Java technologies for real-time and embedded systems (JTRES), 2007.

[29] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, *Region-based memory management in cyclone*, in Conference on Programming Language Design and Implementation (PLDI), 2002.

[30] R. Henriksson, *Scheduling Garbage Colection in Embedded Systems*, PhD thesis, Lund University, July 1998.

[31] M. Hertz and E. D. Berger, *Quantifying the performance of garbage collection vs. explicit memory management*, in Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005.

[32] R. L. Hudson and J. E. B. Moss, *Sapphire: Copying GC without stopping the world*, in ISCOPE Conference, 2001.

[33] IBM, *DDG1000 Next Generation Navy Destroyers,* www.ibm.com/-press/us/en/pressrelease/21033.wss, 2007.

[34] JSR 302, *Safety critical Java technology*, 2007.

[35] N. Juillerat, S. Müller Arisona, and S. Schubiger-Banz, *Real-time, low latency audio processing in java*, in International Computer Music Conference, 2007.

[36] H. Lieberman and C. Hewitt, *A real-time garbage collector based on the lifetimes of objects*, Communications of the ACM, 26, 1983.

[37] T. F. Lim, P. Pardyak, and B. N. Bershad, *A memory-efficient real-time non-copying garbage collector*, in International Symposium on Memory Management (ISMM), 1998.

[38] S. Meyers, *More Effective C++*, Addison-Wesley, 1997.

[39] B. Milewski, *C++ In Action: Industrial-Strength Programming Techniques*, Addison-Wesley, 2001.

[40] S. Nettles and J. O'Toole, *Real-time replication garbage collection*, in Conference on Programming Language Design and Implementation (PLDI), 1993.

[41] A. F. Niessner and E. G. Benowitz, *Rtsj memory areas and their affects on the performance of a flight-like attitude control system*, in International Workshop on Java technologies for real-time and embedded systems (JTRES), 2003.

[42] K. D. Nilsen, *Garbage collection of strings and linked data structured in real time*, Software, Practice & Experience, 18, 1988.

[43] J. Noble, J. Potter, and J. Vitek, *Flexible alias protection*, in European Conference on Object-Oriented Programming (ECOOP), 1998.

[44] K. Palacz and J. Vitek, *Java subtype tests in real-time*, in European Conference on Object-Oriented Programming (ECOOP), 2003.

[45] P. P. Pirinen, *Barrier techniques for incremental tracing.*, in International Symposium on Memory Management (ISMM), 1998.

[46] F. Pizlo, J. Fox, D. Holmes, and J. Vitek, *Real-time Java scoped memory: design patterns and semantics*, in International Symposium on Object-oriented Real-Time Distributed Computing (ISORC), 2004.

[47] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard, *Stopless: A real-time garbage collector for modern platforms*, in International Symposium on Memory Management (ISMM), 2007.

[48] F. Pizlo, E. Petrank, and B. Steensgaard, *A Study of Concurrent Real-Time Garbage Collectors*, in Programming Language Design and Implementation (PLDI), 2008.

[49] F. Pizlo, A. Hosking, and J. Vitek, *Hiearchical real-time garbage collection*, in Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2007.

[50] F. Pizlo and J. Vitek, *An empirical evalutation of memory management alternatives for Real-time Java*, in Real-Time Systems Symposium (RTSS), Dec. 2006.

[51] Purdue, *The Ovm virtual machine,* www.ovmj.org.

[52] W. J. Schmidt and K. D. Nilsen, *Performance of a hardware-assisted real-time garbage collector*, in Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[53] F. Siebert, *Real-time garbage collection in multi-threaded systems on a single processor*, in Real-Time Systems Symposium (RTSS), 1999.

[54] F. Siebert, *Constant-time root scanning for deterministic garbage collection.*, in International Conference on Compiler Construction (CC), 2001.

[55] F. Siebert, *The impact of realtime garbage collection on realtime Java programming.*, in International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), 2004.

[56] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove, *Eventrons: a safe programming construct for high-frequency hard real-time applications*, in Conference on Programming language design and implementation (PLI), 2006.

[57] J. Spring, F. Pizlo, R. Guerraoui, and J. Vitek, *Reflexes: Abstractions for highly responsive systems*, in International Conference on Virtual Execution Environments (VEE), 2007.

[58] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, *StreamFlex: High-throughput stream programming in Java*, in Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 2007.

[59] Sun Microsystems, *Sun java real-time system,* http://java.sun.com/javase/technologies/realtime/, 2008.

[60] J.-P. Talpin and P. Jouvelot, *Polymorphic type, region, and effect inference*, Journal of Functional Programming, 2, 1992.

[61] M. Tofte and J.-P. Talpin, *Region-Based Memory Management*, Information and Computation, 132, 1997.

[62] T. Yuasa, *Real-time garbage collection on general-purpose machines*, Journal of Systems and Software, 11, 1990.

[63] T. Zhao, J. Noble, and J. Vitek, *Scoped types for real-time Java*, in IEEE International Real-Time Systems Symposium (RTSS), 2004.