The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems

David F. Bacon, Perry Cheng, and V.T. Rajan

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.
dfb@watson.ibm.com
{perryche,vtrajan}@us.ibm.com

Abstract. With the wide-spread adoption of Java, there is significant interest in using the language for programming real-time systems. The community has generally viewed a truly real-time garbage collector as being impossible to build, and has instead focused its efforts on adding manual memory management mechanisms to Java. Unfortunately, these mechanisms are an awkward fit for the language: they introduce significant run-time overhead, introduce run-time memory access exceptions, and greatly complicate the development of library code. In recent work we have shown that it is possible to build a real-time collector for Java with highly regular CPU utilization and greatly reduced memory footprint. The system currently achieves 6 ms pause times with 50% CPU utilization (MMU) and virtually no "tail" in the distribution. We show how this work can be incorporated into a general real-time framework, and extended to systems with higher task frequencies. We argue that the community should focus more effort on such a simple, orthogonal solution that is true to the spirit of the Java language.

1 Introduction

Garbage collected languages like Java are making significant inroads into domains with hard real-time concerns, such as automotive command-and-control systems. However, the engineering and product life-cycle advantages consequent from the simplicity of programming with garbage collection remain unavailable for use in the core functionality of such systems, where hard real-time constraints must be met. As a result, real-time programming requires the use of multiple languages, or at least (in the case of the Real-Time Specification for Java [4], or RTSJ) two programming models within the same language. Therefore, there is a pressing practical need for a system that can provide real-time guarantees for Java without imposing major penalties in space or time.

In previous work [2,1], we presented the design and evaluation of a uniprocessor collector that is able to achieve high CPU utilization during collection with far less memory overhead than previous real-time garbage collectors, and that is able to guarantee time and space bounds provided that the application can be accurately characterized in terms of its maximum live memory and average allocation rate over a collection interval.

In this position paper, we begin by describing the weakness of the programming model introduced by the RTSJ, both in terms of usability by the programmer and in terms of burdens it places on the virtual machine.

R. Meersman and Z. Tari (Eds.): OTM Workshops 2003, LNCS 2889, pp. 466–478, 2003.
© Springer-Verlag Berlin Heidelberg 2003

We then provide a brief overview of the features of our collector, describe how it can be applied to create a far simpler real-time programming interface, and discuss how to improve its resolution so that it can be used to program systems that require reponse times in the tens of microseconds.

2 Problems with RTSJ

The Real-Time Specifiaction for Java (RTSJ) is a standard for extending Java to meet the needs of real-time applications. The specification identifies seven areas of interest. The two areas relevant to this paper are thread scheduling and memory management. The design of RTSJ memory management is heavily influenced by a desire to "allow the allocation and reclamation of objects outside of any interference by any GC algorithm". This policy arose from the expert group's belief that garbage collection alone could not sufficiently meet real-time needs. Instead, real-time threads (NoHeapRealTimeThreads) can allocate and manipulate objects only from two new memory areas (the immortal heap and manually programmed scoped memory regions) which are free from GC interference. The (regular) heap is still managed by the garbage collection. To maintain the separation between these regimes, there are restrictions on references between objects from different memory regions. The goal is to enable the collection of scoped memory regions independent of the heap and for the real-time threads to always be able to pre-empt the GC thread.

In this section, we will examine the aspects of the RTSJ design that relate to memory management and evaluate the design's effectiveness in providing an overall real-time solution to the problem of memory management.

2.1 Description

RTSJ provides two additional types of real-time threads. Running at the lowest priorty are traditional Java threads which are subject to the pauses introduced by the garbage collector. RTSJ's RealTimeThread run at a higher priority than the garbage collector but because it can access the heap, cannot arbitrarily pre-empt the GC. Instead it must be delayed for up to the GC-induced latency. The standard does not require this latency to be low. In the case that a stop-the-world collector is used, a RealTimeThread is no better than a regular thread. Finally, RTSJ's NoHeapRealTimeThread can pre-empt the GC at any moment and any encountered latency is due to the cost of context switching and scheduler computations. To support this, it is illegal for such threads to manipulate or refer to any object in the heap.

The severe restrictions on NoHeapRealTimeThread is ameliorated by the introduction of additional memory areas. We focus on the immortal heap and scoped memory regions. Objects allocated into the immortal heap have lifetimes for the remaining duration of the program. It is possible for objects allocated in the immortal heap to become inaccessible but they remain alive simply by virtue of residing in this region. Secondly, scoped memory regions support objects whose lifetimes are shorter in duration than the duration of the entire application and follow a LIFO pattern. A scoped memory region supports memory allocation but does not necessarily support garbage collection. The

scoped memory's size is chosen at its creation time. The set of all scoped memory regions form a tree with the immortal region and the heap as the implicit root. This graph is dynamically implied by the stack-like order in which different threads enter and exit these scoped regions. A thread will, by default, allocate objects in the scope it is currently in. Objects residing in scoped regions can only refer to objects residing in an outer scope (i.e., an ancestral scope). Since the immortal region and the heap form the root node, objects in a scope can refer to an object in the immortal region or the heap, but not vice versa. The overall effect of the scopes is that once all threads leave a scope, the entire scoped memory region can be recycled without tracing through the region.

2.2 Barriers

To enforce the pointer restrictions above, RTSJ uses runtime checks that will throw either an IllegalAssignmentError or a MemoryAccessError if an operation is about to violate the conditions. Specifically, whenever a reference is about to be loaded, a *read barrier* will throw the MemoryAccessError exception if the executing thread is a NoHeapRealtimeThread and the loaded value resides in the heap. Secondly, whenever object X is being stored into object Y, a *write barrier* will throw the IllegalAssignmentError if the scope of X is not an outer scope of the scope of Y [7].

This particular read barrier is hard to optimize. First, coalescing nearby read barriers on the same object is difficult because the barrier is field-dependent. That is, if different fields of an object X are accessed, the read barrier must be applied repeatedly to each access since the exception is dependent not on where X resides but on the contents of the fields. Second, since a method can be executed by both regular threads and NoHeapRealTimeThreads, the read barrier must check the thread type at each barrier. Even coalescing thread checks within a single method is difficult without affecting the other part of barrier.

The write barrier suffers from two factors. Even if the source and target objects of the assignment are not in scoped regions, a dynamic test to ensure this is required. In cases where the objects are in scoped areas, the test must first determine the object's scopes and then determine whether one scope is an outer scope of another scope. The overall write barrier will likely include several memory operations and several conditional instructions.

It is worth emphasizing that these barriers are imposed by the RTSJ and are entirely separate from the barriers, if any, that a particular garbage collector might impose.

2.3 Difficult Usage

In RTSJ, only the NoHeapRealTimeThread is guaranteed true pre-emption. However, it is unclear how to program such a thread if it needs to allocate objects whose lifetimes are unclear. That is, objects that are neither immortal nor follow the LIFO pattern of scoped regions. However, RTSJ does provide wait-free queues that allow real-time thread to safely synchronize with other threads without priority inversion problems.

Consider a real-time server where the high-priority NoHeapRealTimeThread handles incoming queries and sends the request to a lower priorty regular thread via a wait-free queue. Assuming that the request is a String object, choosing where to allocate the

String object is problematic. Clearly the object is not immortal and allocating it from the immortal region will lead to a memory leak if the server is up for any appreciable time. If one were to allocate the object in a scoped region, the leakage problem becomes when the scope would be exited because it is possible that the scope would always contain at least the most recent request. It is probably possible to overcome this scenario by appropriate synchronization on when to enter and exit scopes along with data-copying. However, it seems clear there are programming patterns (for example, FIFO) that are inexpressable or hard to express with scoped memory regions.

Aside from the problem of expressability, there is also the pragmatic problem of existing library code. The vast majority of library code does not use RTSJ features and therefore allocate objects in the regular heap. As a result, any NoHeapRealTimeThread that uses the library code will result in an exception being thrown at run-time.

2.4 Fragmentation

The separation of memory into scoped regions also burdens the programmer with determining the maximum size of the each scoped memory area necessary for execution. In contrast, a regular Java program needs to determine only a single parameter for the heap size. The need to determine memory usage in such a fine-grained manner may require over-provisioning. Consider a thread that enters an outer scope A and an inner scope B and will allocate a total of 100 KB among the two scopes. If the distribution of the 100 objects is unknown until the scopes are entered, then both scopes must be able to accomodate 100 KB, resulting in a total memory usage of 200 KB. On the other hand, allocating the objects in the heap would require only 100 KB. This overhead is above and beyond the wastage associated with not garbage collecting a region.

3 Overview of the Metronome

We begin by summarizing the results of our previous work [2,1] and describing the algorithm and engineering of the collector in sufficient detail to serve as a basis for understanding the work described in this paper.

Our collector, the Metronome, is an incremental uni-processor collector targeted at embedded systems. It uses a hybrid approach of non-copying mark-sweep (in the common case) and copying collection (when fragmentation occurs).

The collector is a snapshot-at-the-beginning algorithm that allocates objects black (marked). While it has been argued that such a collector can increase floating garbage, the worst-case performance is no different from other approaches and the termination condition is easier to enforce. Other real-time collectors have used a similar approach.

Figures 1 and 2 show the real-time performance of our collector. Unlike previous real-time collectors, there is no "tail" in the distribution of pause times, CPU utilization remains very close to the target, and memory overhead is low — comparable to the requirements of stop-the-world collectors. In this section we explain how the Metronome achieves these goals.

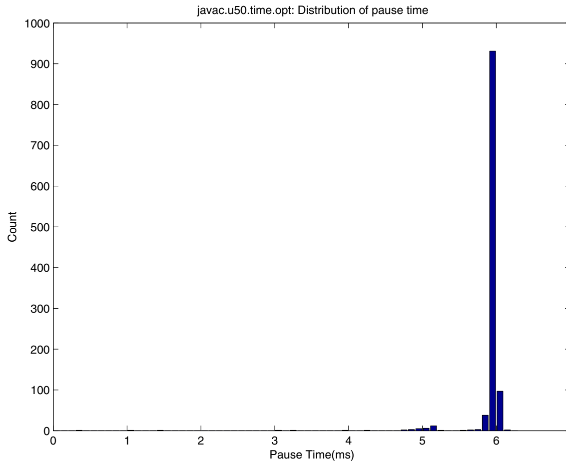

Fig. 1. Pause time distributions for javac in the Metronome, with target maximum pause time of 6 ms. Note the absence of a "tail" after the target time.

3.1 Features of Our Collector

Our collector is based on the following principles:

Segregated Free Lists. Allocation is performed using segregated free lists. Memory is divided into fixed-sized pages, and each page is divided into blocks of a particular size. Objects are allocated from the smallest size class that can contain the object.

Mostly Non-copying. Since fragmentation is rare, objects are usually not moved.

Defragmentation. If a page becomes fragmented due to garbage collection, its objects are moved to another (mostly full) page.

Read Barrier. Relocation of objects is achieved by using a forwarding pointer located in the header of each object [5]. A read barrier maintains a to-space invariant (mutators always see objects in the to-space).

Incremental Mark-Sweep. Collection is a standard incremental mark-sweep similar to Yuasa's snapshot-at-the-beginning algorithm [8] implemented with a weak tri-color invariant. We extend traversal during marking so that it redirects any pointers pointing at from-space so they point at to-space. Therefore, at the end of a marking phase, the relocated objects of the previous collection can be freed.

Arraylets. Large arrays are broken into fixed-size pieces (which we call arraylets) to bound the work of scanning or copying an array and to bound external fragmentation caused by large objects.

Since our collector is not concurrent, we explicitly control the interleaving of the mutator and the collector. We use the term *collection* to refer to a complete mark/sweep/defragment cycle and the term *collector quantum* to refer to a scheduler quantum in which the collector runs.

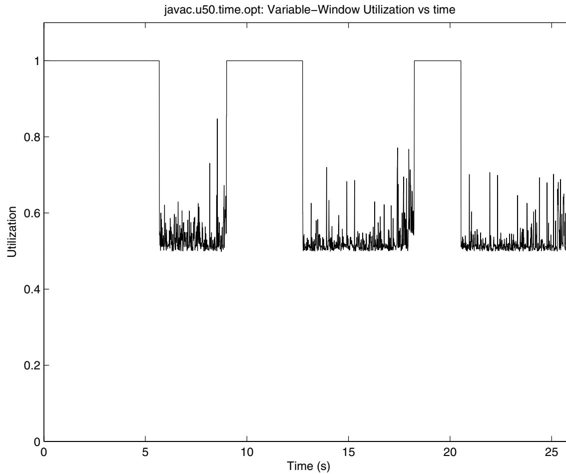

Fig. 2. CPU utilization for javac under the Metronome. Mutator interval is 6 ms, collector interval is 6 ms, for an overall utilization target of 50%; the collector achieves this within 3% variation.

3.2 Read Barrier

We use a Brooks-style read barrier [5]: each object contains a forwarding pointer that normally points to itself, but when the object has been moved, points to the moved object.

Our collector thus maintains a to-space invariant: the mutator always sees the new version of an object. However, the sets comprising from-space and to-space have a large intersection, rather than being completely disjoint as in a pure copying collector.

While we use a read barrier and a to-space invariant, our collector does not suffer from variations in mutator utilization because all of the work of finding and moving objects is performed by the collector.

Read barriers, especially when implemented in software, are frequently avoided because they are considered to be too costly. We have shown that this is not the case when they are implemented carefully in an optimizing compiler and the compiler is able to optimize the barriers.

We apply a number of optimizations to reduce the cost of read barriers, including well-known optimizations like common subexpression elimination, as well as special-purpose optimizations like barrier-sinking, in which we sink the barrier down to its point of use, which allows the null-check required by the Java object dereference to be folded into the null-check required by the barrier (since the pointer can be null, the barrier can not perform the forwarding unconditionally).

This optimization works with whatever null-checking approach is used by the run-time system, whether via explicit comparisons or implicit traps on null dereferences. The important point is that we usually avoid introducing extra explicit checks for null, and we guarantee that any exception due to a null pointer occurs at the same place as it would have in the original program.

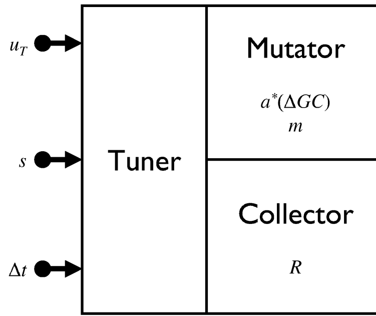

Fig. 3. Tuning the performance of an application (mutator) with the collector. The mutator and collector each have certain intrinsic properties (for the mutator, the allocation rate over the time interval of a collection, and the maximum live memory usage; for the collector, the rate at which memory can be traced). In addition, the user can select, at a given time resolution, either the utilization or the space bound (the other parameter will be dependent).

The result of our optimizations is that for the SPECjvm98 benchmarks, read barriers only have a mean cost of only 4%, or 9.6% in the worst case (in the `201.compress` benchmark).

3.3 Time-Based Scheduling

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [3], has used work-based scheduling. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The reason for this is simple: work-based algorithms do a little bit of collection work each time the mutator allocates memory. The idea is that by keeping this interruption short, the work of collection will naturally be spread evenly throughout the application. Unfortunately, programs are not uniform in their allocation behavior over short time scales; rather, they are bursty. As a result, work-based strategies suffer from very poor mutator utilization during such bursts of allocation.

In fact, we showed both analytically and experimentally that work-based collectors are subject to these problems and that utilization often drops to 0 at real-time intervals.

Time-based scheduling simply interleaves the collector and the mutator on a fixed schedule. While there has been concern that time-based systems may be subject to space explosion, we have shown that in fact they are quite stable, and only require a small number of coarse parameters that describe the application's memory characteristics in order to function within well-controlled space bounds.

3.4 Provable Real-Time Bounds

Our collector achieves guaranteed performance provided the application is correctly characterized by the user. In particular, the user must be able to specify the maximum

amount of simultaneously live data m as well as the peak allocation rate over the time interval of a garbage collection $a^\star(\Delta GC)$. The collector is parameterized by its tracing rate R.

Given these characteristics of the mutator and the collector, the user then has the ability to tune the performance of the system using three inter-related parameters: total memory consumption s, minimum guaranteed CPU utilization u_T, and the resolution at which the utilization is calculated Δt.

The relationship between these parameters is shown graphically in Figure 3. The mutator is characterized by its allocation rate over the interval of a garbage collection $a^\star(\Delta GC)$ and by its maximum memory requirement m. The collector is characterized by its collection rate R. The tunable parameters are Δt, the frequency at which the collector is scheduled, and either the CPU utilization level of the application u_T (in which case a memory size s is determined), or a memory size s which determines the utilization level u_T.

Note that in either case both space and time bounds are guaranteed.

4 Integrating the Metronome with a Real-Time System

As we showed in Section 2, the RTSJ treats garbage collection as a foreign entity, outside of the normal scheduling and priority mechanisms of the system. This in turn leads to the requirement to create various different types of memory regions, with complex restrictions on which regions and thread types can reference other regions. The end result is a system which lacks orthogonality, introduces unpredictable run-time exceptions, and makes development and understanding of library code extremely difficult.

We advocate a different approach: integrating collection into the run-time system, and particularly, into the scheduler, in such a way that garbage collection is a real-time task like all others.

The benefits of this approach are enormous in terms of simplification of the programming model. Since one of the major benefits of Java is its reliability and simplicity, we believe this is fundamental to the spirit of a real-time implementation of Java.

A model based on a truly real-time collector is simpler in the following ways:

- only a single memory space;
- no run-time exceptions on memory accesses;
- ability to share objects between real-time and non-real-time threads; and
- ability of real-time threads to call standard library routines.

In the previous section we presented an overview of the Metronome and how it schedules garbage collection. This scheduling approach can easily be adapted to periodic real-time scheduling. The interval Δt is the period of the collector. The utilization u_T is the fraction of that time devoted to the collector. The user parameterizes the application in terms of its allocation rate, which is already an RTSJ parameter on real-time threads. The one additional required parameter is the maximum memory utilization m.

With these parameters, the garbage collector becomes a periodic real-time task. The time remaining after garbage collection, $1 - u_T$, is the time available in which to

schedule the high-priority real-time tasks. A feasible schedule must be able to perform the real-time tasks in this interval.

Of course, the main limitation of this approach is that collection can consume a significant portion of the total processor resources. In our experiments we have used the SPECjvm98 benchmarks as a driving workload for our collector, effectively treating them as high-priority processes. The result is that when collection is on, it consumes about 50% of CPU resources.

While this seems high, there are two important points to note: first of all, the domain in which Java is likely to prosper is one in which the greater concerns are with development time and reliability, and less with CPU cost. A purely garbage-collection based real-time environment should have significant time-to-market advantages over the much more complex model of the RTSJ.

The second point is that the high-priority tasks are likely to have a much lower allocation rate, or could be programmed to do so. In that case, the percentage of the CPU that has to be allocated to the collector will significantly decrease.

Of course, this begs the question: are we better off with a simple programming model in which programmers have to adapt by reducing the allocation rate of some performance-critical code, or with a more complex programming model that gives them some tools (like scoped memory regions) for reducing this allocation rate. We believe that a simple, uniform, adaptable programming model is preferable.

5 Reducing Context Switch Times

The Metronome currently operates with a maximum pause time of 4 milliseconds, and using the current approaches we expect to drive this pause time to the sub-millisecond level. However, for some applications responses in the tens of microseconds are required. In this section we describe the features of the current collector that stand in the way of this goal, and describe how the design could be adapted to achieve these much lower pause times.

5.1 Priority Scheduling

First of all, the current system does not include any notion of high-priority real-time threads versus low-priority threads. Such a distinction would have to be incorporated, with the collector having a priority higher than the low-priority threads but lower than the high-priority threads. When scheduling the high-priority threads, a feasible schedule would have to include a time allotment for the collector thread to run, that was sufficient given the application threads' cumulative allocation rates.

This could easiy be done by treating the addition of a thread with allocation rate a_i and maximum live memory m_i as the addition of two threads to the schedule: one is the thread itself, and the other is the additional work performed by the garbage collector.

High-priority real-time threads would be allowed to interrupt the collector thread, but the scheduling algorithm guarantee guarantee that the collector receives sufficient resources in each time period.

The modifications described below will allow much faster interruption of the collector.

5.2 Lazy Read Barrier

The first inhibitor to quick context switching out of the collector is an optimization of the read barrier which we call the “eager barrier”. In this form of the barrier, when a mutator thread loads a pointer onto the stack its forwarding pointer is immediately followed. This has the advantage that if the pointer is used in a loop, the read barrier is only executed once. However, it does mean that if the mutator is interleaved with the collector, and the collector moves objects, it must execute “fix-up” code on the stack frames to maintain the eager invariant (that is, that all stack references point to the current versions of objects).

By trading off some throughput for response time, we can employ the lazy version of the read barrier, which does not forward the pointer until it is used. In this case, there is no fix-up code required and any movement of objects by the collector does not inhibit context switch to the mutator. We measured the cost of this as about a 2% slowdown over the SPECjvm98 benchmarks.

Based on our experience we believe that the performance loss due to the lazy barrier can mostly be recovered, albeit at the expense of more complex compiler optimizations. Essentially, there is a spectrum between “eager” and “lazy”, and compiler optimizations can preserve the lazy property while reducing the number of forwarding operations.

5.3 Abortable Copy Operations

The next problem for context switching out of the collector is that it may be in the midst of a long-running atomic operation, in particular copying a large chunk of memory like a stacklet or an arraylet.

The solution is to make these operations abortable and restartable. The main difficulty is that the context switches must not be so frequent that the abort operations are able to impede forward progress of the collector. Thus the cost of the aborted operations must be factored into the collector cost.

5.4 Deferred Root Scanning

Probably the single largest inhibitor to rapid context switching out of the collector is the atomic operations performed when collection starts. In particular, the collector uses a “snapshot-at-the-beginning” technique. Thus, the roots in the thread stacks and global variables must be copied atomically.

Stacklets [6] allow the delay to be reduced by only requiring the snapshoting of the topmost stacklet of each thread, but this solution does not scale to large numbers of threads and introduces further problems because it requires that threads perform stack snapshot operations if they return from the topmost stacklet into a lower, un-snapshotted stacklet. Neither of these behaviors is acceptable in a high-frequency real-time environment.

The solution is to weaken the “snapshot-at-the-beginning” property. Instead we simply require that no reference from a stack be allowed to “escape” from the stack without being logged. Thus the write barrier, instead of recording just the old value (the Yuasa-style barrier) also records new values written from stack variables into the heap (the

Dijkstra-style barrier). In this manner, no references on the stacks can escape into the heap without being caught by the write barrier.

As a result, we both avoid the termination problems of the Dijkstra-style barrier as well as avoiding the need to complicate the read barrier (for instance by recording pointers loaded onto the stack during collection). Furthermore, even though the stacks may be changed by the mutators, we still only have to scan each stack exactly once (and can do so incrementally), since any relevant references not found in the stack scan must have been written by the write barrier.

This in turn allows us to interleave execution of mutators with root scanning, at a modest performance cost in the write barrier.

5.5 Safe Points

Finally, since our scheduler only performs context switches at safepoints, there is the issue of delay introduced while waiting for threads to reach a safe point. In practice, safe points occur quite often. In order to meet real-time bounds, an analysis phase can be added which inserts extra safe points into large monolithic basic blocks.

6 Metronome versus RTSJ

In Section 2, we described some of the problems with RTSJ. Now that we have presented the Metronome, we examine the relative benefits of the two approaches.

The current collector can usually context switch in about 100 microseconds but in certain (short) phases of collection may take as much as 700 microseconds. With the modifications of Section 5, in particular the lazy read barrier and deferred root scanning, we expect to be able to bound the context switch time to 100 microseconds.

In choosing between RTSJ and the Metronome, one must balance greater control over memory usage and possibly superior performance with the greater programming effort. Metronome has the advantage of greater simplicity and retains the spirit of Java where the programmer is not burdened with memory management details: there are no scopes and no dynamic memory store or load exceptions.

RTSJ has the advantage that it may be possible to write a particular program to fit in a much smaller memory by careful use of ScopedMemory. However, using ScopedMemory might actually increase memory consumption because the size estimate is too conservative or if a large fraction of data within the scope dies during the scope's lifetime. RTSJ also has the advantage that context switching to NoHeapRealTimeThreads can be quicker than context switching out of the Metronome.

To capture the key benefits of both systems, we propose a hybrid system in which RTSJ is modified to take real-time GC into consideration. We propose removing the NoHeapRealtimeThread class so that the programmers do not have to program in a constrained fashion in which high-priority threads cannot access the heap at all. Instead, high-priority threads can communicate with lower-priorty or even regular threads in the usual way. In the hybrid system, MemoryAccessError exceptions are eliminated because the read barriers associated with the NoHeapRealtimeThread are removed.

In this hybrid system, the expected development cycle begins with programming without scoped memory except where the fit is natural and intuitive. In prototyping the system, the programmer must then determine the memory characteristics of the program by some combination of analysis and profiling. These will result in establishing the number and size of scoped regions, the maximum live heap data, and the allocation rates of the various threads. From these parameters, the overall computational and space requirements of the garbage collector can be established and the feasibility of the entire system can be determined. Should the heap pose a problem, the programmer must reduce the allocation rate and live heap data by modifying the program logic or by increasing the use of scoped memory regions. What distinguishes this hybrid system from RTSJ is a less brittle programming model where memory requirements can be met by incrementally tightening the memory usage of the program. This is made possible by a greater reliance on the garbage collector. As mentioned before, the floating garbage possible with scopes can make scoped memory less attractive than garbage collection.

7 Conclusions

We have described the complexities of the RTSJ programming model, and shown that they will have an adverse effect on both ease of use and reliability, and may have adverse performance effects as well.

We have proposed an alternate approach to creating a real-time Java programming environment, which is based on constructing a true real-time garbage collector which is fully integrated with the scheduling system. This allows garbage collection to co-exist, and results in a much simpler programming model.

We believe that such a model is more consistent with the spirit of the Java language and will ultimately be more useful to the potential body of real-time Java programmers.

The techniques of the Metronome can also be applied to simplify the RTSJ specification by eliminating the need for threads that have no references to the heap, and eliminating an entire class of run-time memory access exceptions.

References

[1] Bacon, D. F., Cheng, P., and Rajan, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, June 2003).

[2] Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003), pp. 285–298.

[3] Baker, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

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

[5] Brooks, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.

[6] Cheng, P., Harper, R., and Lee, P. Generational stack collection and profile-driven pretenuring. In *Proc. of the Conference on Programming Language Design and Implementation* (June 1998). *SIGPLAN Notices*, *33*, 6, 162–173.
[7] Higuera-Toledano, M. T., and Issarny, V. Analyzing the performance of memory management in RTSJ. In *The 5th IEEE International Symposim on Object-oriented Real-time Distributed Computing* (Crystal City, Virginia, 2002).
[8] Yuasa, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.