

Appeared in Proceedings of 7th IEEE International Symposium on Object-oriented Real-time distributed Computing

(ISORC 2004), Vienna, Austria, May 2004.

A Hard Look at Hard Real-Time Garbage Collection

David Detlefs

Sun Microsystems

1 Network Drive, Burlington, MA 01803

david.detlefs@sun.com

Abstract

In this paper, I review the literature on the use of garbage collection in real-time systems. I concentrate on hard real-time systems, where we ideally construct mathematical proofs of correctness and of timing properties. In particular, I examine the interaction of overheads imposed on mutator operations by garbage collection algorithms on worst-case execution time analyses of real-time threads performing those operations. In recent years there has been a shift from work-based to time-based approaches. This paper explains and motivates this shift, and reviews examples, problems, and advantages of example algorithms from each approach. Finally, I examine what extensions to programming verification technology might be necessary to prove that sufficient memory space exists to run a real-time system with the same rigor that one proves that sufficient time exists in a real-time schedule.

1. Introduction

The last decade has seen the widespread commercial adoption of programming languages such as the Java™ programming language and later Microsoft's C# language. Programmers in these languages report greater productivity than with predecessor languages; there is widespread agreement that one of the major productivity-enhancing features of these languages is the use of automatic storage management, or garbage collection (or just GC).

The use of the Java programming language spans a wide spectrum of applications, from large commercial systems running on powerful server machines to cell phones. One of the original targets of the language design was real-time systems. However, some features of the language were not compatible with real-time programming: the use of garbage collection was perhaps foremost. Therefore, the *Real Time*

Specification for Java (henceforth *RTSJ*) [?] was designed. This is an extension of Java that allows real-time programming. A major aspect of the RTSJ design is the requirement that real-time threads not allocate memory in the garbage-collected heap. Instead, they are required to use *scoped memory regions*, in which allocation can occur in constant time, and which can be deallocated in their entirety in constant time.

Scoped memory is a form of manual memory management, and is subject to many of the dangers inherent in manual memory management. Memory scopes form a hierarchy; objects in child scopes may point to objects in parent scopes, but not vice versa. Attempts to create a parent-to-child pointer are required to generate a run-time exception. (The “regular” heap is viewed as the parent of all other scopes in the RTSJ.) While the motivations for these constructs in real-time programming are understandable, they also complicate the language by introducing different classes of storage, and a new kind of run-time error whose absence must be proved. There has been considerable interest, therefore, in whether it is possible to restore the original simplicity of the language by implementing a garbage collector that supports real-time programming. (Obviously, there was also considerable interest in garbage collection for real-time systems before the advent of the Java programming language.) The goal is to do away with the memory-management section of the RTSJ, allowing real-time programs to use heap-allocated objects freely, with arbitrary pointer relationships. Of course, the obligation to prove that the program plus garbage collector meets the system's real-time constraints remains, as we will discuss.

In this paper I survey the current state of the art in real-time GC, paying particular attention to recent work suggesting that collection should be scheduled as one of the real-time activities. In a later section, I consider what sort of program analyses would be required if one wanted to mathematically prove that a program did not violate real-time constraints because of garbage collection.

2. Early work in real-time garbage collection.

One of the earliest papers discussing the design of a real-time collector is Henry Baker’s incremental copying collector [?]. This work is notable not only for the specific contribution of its technique, but also because it established a model for the kind of analysis that would allow a collector to be considered “real-time.” Baker started from the existing idea of *semispace copying* garbage collection [?]. In this technique, available memory is partitioned into two equal *semispaces*, labeled *from-space* and *to-space*. Allocation proceeds in from-space until it is full, at which point a collection occurs. The live objects in from-space are copied to *to-space*. From-space is now free; usually only some fraction of to-space is filled with copied objects. The two semispaces then exchange roles, so allocation again precedes in (the new) from-space, until it is filled and the process repeats in the other direction. Baker cleverly modified this algorithm to interleave collector and *mutator* (i.e., the rest of the program, from the garbage collector’s viewpoint) operations, in such a way that the collection appears semantically to happen atomically in a single short *flip* operation. Every mutator read operation performs a *read barrier* to preserve the illusion that the collection happened instantaneously: if the read is through a pointer to a from-space object, the object is first copied to to-space. In addition, each mutator allocation also performs some bounded amount of garbage collection work. A parameter k controls the amount of collection work performed per allocation; larger values of k cause collection to finish quickly after the flip, but also cause mutator operation to “slow down” more during collection.

In the next section we will discuss whether the Baker approach really succeeds in creating a collector suitable for use in hard real-time systems. It should be noted, however, that this was probably not Baker’s real motivation. Instead, by paying attention to real-time concerns he created a collector that decreased, in an informal sense, the “intrusiveness” of garbage collection in interactive applications. His goal, and approach, were followed by a number of subsequent papers in a similar vein: for example, Brooks [?] had a clever variant on Baker’s collector that moved some of the GC overhead from the generally more frequent *read barriers* to less frequent *write barriers*. A technique used in Brooks’ collector has been important in some more recent real-time collectors; the technique and these collectors are described in section ???. The Baker style of real-time GC analysis is still used. For example, the collector of Cheng and Blleloch [?], which is a parallel extension of the *replicating collection* idea of O’Toole and Nettles [?], uses the Baker methodology to show that the collector is real-time.

The properties of these collectors are summarized in table ??, which is described in section ??.

3. Problems with work-based real-time collectors.

In this section I will discuss problems with using the style of real-time collector initiated by Baker in formal hard real-time systems. I will note up front that this section is largely a restatement of arguments by Henriksson [?] and in the *Metronome* work of Bacon, Cheng, and Rajan [?]. However, the alternative approach they advocate is one that our group has also been developing for several years, albeit for soft-real time applications. In section ?? I will describe our work in this area.

Are work-based real-time collectors in the Baker style sufficient to allow the use of GC in hard real-time systems? I think the answer is “yes” in a strict formal sense, but this answer may not be of practical use: the more conservative worst-case analysis that is required may lead to bounds so pessimistic that almost no real-time schedules can be satisfied. I will explain this point further in the rest of this section.

I take the following as the definition of hard real-time systems: there are some set of input events that may occur and require the system to produce some corresponding output event in a timely manner. (Some of these events may just be timer expiration notifications.) For example, a periodic event that measures an aircraft’s attitude may require signals to be sent within 20 ms to produce corresponding changes in the settings of the aircraft’s control surfaces. So the input to the scheduling problem is some set of events, with each event specifying an upper bound on its inter-arrival times, and an upper bound on the permitted time for producing the corresponding output event. For a given set of possible input events, and a program to process them, we can generate a set of real-time constraints (at least x ms of every y must be devoted to servicing events of type E), and scheduling theory tells us whether a set of such constraints can be simultaneously satisfied.

If one takes hard real-time seriously (as I hope those designing safety-critical real-time systems do!), then one has to perform careful program analysis to determine worst-case execution time bounds on all program paths that could be taken in performing a task. Worst-case bounds mean assuming worst-case bounds on all the constituent operations – this is why we often hear that cache memory is of no use in a hard real-time system, since the analysis must assume that the cache misses anyway. The approach of the Baker collector is to alter this analysis by assuming larger, but still bounded, worst-case times for operations that may have associated collector activity (i.e., reads and allocations for the Baker collector in particular). Bacon, Cheng, and Rajan call this style *work-based* garbage collection scheduling. In this scheduling style, the collector is invisible to the real-time scheduling; it is a general “tax” exacted on mutator opera-

tions.

I will examine the effect of the Baker collector on worst-case execution times in detail. Reads of pointer fields of heap objects may incur the additional cost of copying one additional object. Baker, working in a Lisp system, assumed a heap filled with fixed-size `cons` cells in his presentation, so this is a bounded cost. He sketches other schemes for copying variable sized objects while preserving a bound. Allocation may cause some fixed amount of associated collection work, and may also invoke the flip operation. The flip operation scans the program's *root set*, the locations from which all live objects are reachable (by definition). These may include some set of global variables (whose number is bounded in each particular program), but they also include pointer locations in activation records of the program's threads. These activation records are often on allocated in a stack, and called stack frames; the point is that their number is unbounded, so the cost of the flip operation (as first described in Baker's paper) is unbounded.

There are two kinds of solutions to this problem. One is to treat the program stacks incrementally in the same way updates to heap objects are. This is what Baker proposes: accesses to local variables of methods would be subject to the same barriers as accesses to fields of heap objects. This style of solution is sometimes implemented by allocating a thread's activation records as a linked list in the program heap, so that accesses to method local variables actually *are* accesses to fields of heap objects. The thread stacks themselves may be arranged in another heap data structure, leaving only a single global variable as the root. The drawback of the heap-allocation approach is that activation records of completed invocations must be garbage collected. Global variables may be handled in a similar way – treating an array of global variables as part of the heap. The root set size is then bounded by a constant independent of the program, and the worst-case time of the flip operation is also bounded.

This approach has an important drawback. Accesses to local variables are generally much more frequent than accesses to object fields. (This is why local variables are often stored in hardware registers.) So associating GC operations with local variable accesses has the potential of greatly increasing the worst-case execution time of a code path that accesses locals (as well as increasing average-case costs).

Therefore, another approach has been proposed for this problem, one that tries to bound flip costs while still allowing “normal” access to local variables. This approach divide the stack into fixed size chunks. Cheng and Blleloch [?], who use this approach, call these chunks “stacklets.” Since their collector is in the replicating collector family, it has the equivalent of a flip at the end of collection rather than at the start. To complete the flip, all stack frames must be scanned to convert from-space pointers into to-space point-

ers. They scan stacklets below the top frame incrementally, stopping the mutator only to scan frames in the top stacklet.

I have some worries about the stacklet approach. The collector must not process a stacklet in which a mutator thread is executing, because of concurrency issues. This may be avoided in two ways: either the mutator must be delayed when it attempts to “return into” a stacklet that is being processed until that processing is complete, or the collector's partial processing of a stacklet must be aborted and later retried in this situation. With the latter approach, there is a problem with guaranteeing progress: a thread whose stack height varies frequently by large amounts might repeatedly abort attempts to scan its below-top stacklets. The blocking approach essentially attaches the cost of scanning a stacklet to the worst-case cost of returning from a method invocation. This cost is bounded, but with a relatively large bound.

So, it seems that we can give constant bounds to all the GC work attached to mutator operations in work-based collectors. This allows us to do worst-case execution time analysis of the real time tasks in the normal style, allowing the use of normal real-time scheduling theory.¹ However, the systems may be of limited practical use. The attached GC work increases the costs that must be used in worst-case execution times analysis in ways that may be unrealistic, preventing otherwise-feasible systems from being scheduled. First, it may be that collection finishes quickly when it starts, and that operations seldom need to perform the associated collector work. But we must assume that all operations incur the related cost in the worst-case analysis. A second problem is possible asymmetry in the costs of GC operations potentially associated with mutator operations. For example, the flip operation, which scans thread stacks, may occur on any allocation. This operation (even using techniques like stacklets) may be considerably more expensive than the collection work attached to other allocations. In the worst-case analysis, however, every allocation must be assumed to incur the worst-case cost of the most expensive possible operation.

This is not purely a problem of overly pessimistic analysis. Consider the behavior of the Baker collector, as illustrated in Figure ?? . Before the flip operation, we see occasional allocation (“A”) operations incurring some GC cost. Immediately after the flip operation, however, read (“R”) operations incur GC cost to copy their referents to to-space (since only those objects directly reachable from the stacks were copied in the flip operation). For some period of time

¹ The fact that some GC operations must be atomic keeps the real-time threads from being arbitrarily preemptible, so scheduling models more complicated than “earliest deadline first” must be used. Models that account for jitter from *interference* from higher-priority threads might be appropriate for this problem.

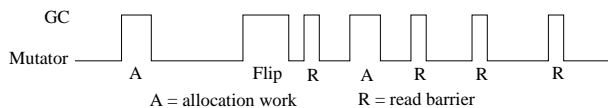


Figure 1. Baker collector behavior after a flip

after the flip, the average-case performance of reads may come close to the theoretical worst-case performance.

As an aside, one possible approach to this problem is to construct a more sophisticated analysis that understands the structure of the garbage collector. For example, flip operations are relatively rare. If several allocations occur along an (atomic) code path, but sufficiently few that they cannot contain two GC cycles, then the analysis could safely conclude that at most one of the allocations incurs the cost of a flip. Martin *et al.* [?] propose an analogous analysis for cache behavior: if a variable is accessed in a loop, it may miss on the first iteration, but might be proved to hit on the second and subsequent iterations. But analyses like these are speculative, and at the very least are considerably more sophisticated than worst case execution time, which is sophisticated enough in its own right.

To summarize: the work-based approach may do a perfectly good job *on average* of spreading GC overhead evenly over mutator operation – it may be fine for applications with “soft” real-time constraints. But when it is applied to hard real-time systems, the large difference between the average costs of the operations and the worst-case costs because of the associated garbage collection actions means that (if worst-case analysis is taken seriously) the actual machine resources may be quite underutilized. The next section presents an alternative approach to real-time collection that corrects this problem.

4. Time-based real-time GC

Henriksson [?], Robertz and Henriksson [?] and Bacon, Cheng, and Rajan [?, ?] all present the kernel of the analysis above, and somewhat similar alternatives. Bacon *et al.* formalize the problem with worked-based approaches, using the *Minimum Mutator Utilization*, or *MMU* metric, defined by Cheng and Blelloch [?]. This is a real-time constraint on collection activity: the mutator must get some guaranteed fraction of every time window of length y .² Such a guarantee is essential to enabling real-time scheduling of the mutator tasks. Bacon *et al.* point out that if one views the collector work in a Baker-style analysis as non-mutator work,

² Though Cheng and Blelloch actually originally used the metric as an *a posteriori* measurement of the “intrusiveness” of their collector, not as an input constraint to the collector.

it is very hard to make any meaningful MMU guarantees – this is another way to state the arguments of the last section. The problem is that the GC and the mutator operations are coupled at too fine a grain: the duration of individual reads or allocations is generally much smaller than the periods and service times of real-time constraints. The motivation for making this coupling in the work-based style of analysis is to leave the real-time scheduling problem for the mutator threads unchanged (except for the fact that the “virtual machine” used in the timing analysis must be considered slower). Henriksson, Robertz, and Bacon *et al.* essentially abandon this advantage. In their alternative *time-based* approach, the garbage collector is treated as another real-time task to be scheduled. While this seems to complicate the problem somewhat, it also gains several advantages. This approach no longer “taxes” individual mutator operations with GC overhead, so the worst-case analysis of mutator code-paths is closer to the actual average, allowing the machine to be more fully utilized. Rare but potentially costly operations such as flip transitions need only be shorter than the intervals during which the GC is scheduled.

What properties must a collector have to permit it to be scheduled as a real-time task? Real-time scheduling theory is simplest when tasks are arbitrarily preemptible, so an ideal collector would be arbitrarily preemptible. As far as I know, no collector achieves this ideal; the next best thing is to have the atomic steps of the collector have worst-cases costs that are as short as possible to minimize “jitter” due to the collector. For example, the collector’s allocation operation must take bounded time, and it must always succeed if sufficient total space is available. I mention these seemingly obvious constraints because they are not satisfied by some heap organizations. In particular, *non-moving* collectors are vulnerable to *fragmentation*: if objects are allocated and deallocated in an unfortunate pattern, a heap may contain more than enough total free space to satisfy an allocation request, but no sufficiently large individual contiguous block. Even without catastrophic failures like this, allocators for non-moving heaps often have bad upper bounds, where some allocations incur dramatically higher cost to coalesce some free blocks, or allocate a large block to be broken into smaller ones of the appropriate size [?]. Large upper bounds on allocation costs, again, decrease the usability of a collection algorithm in a hard-real-time setting. Therefore, I believe that hard real-time systems must perform at least some *compaction*, so that most, if not all, allocation can be in contiguous areas, which has constant cost.

The time-based approach need not result in algorithms radically different from those used in work-based approaches. Henriksson uses a small modification of Brooks’s variant of Baker’s algorithm. The modification is, however, important, because it illustrates the difference in viewpoint. The goal in engineering a time-based real-time col-

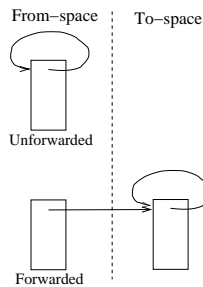


Figure 2. Brooks forwarding pointers

lector is to move as much of the garbage-collection work as possible into the collector's time slices, leaving only the minimum amount of GC work attached to mutator operations. I will first explain Brooks's variant, then Henriksson's modification, as an illustration of this kind of engineering. Baker's correctness proof was based on a *to-space invariant*: after the flip operation, mutator threads observed only to-space objects. This was enforced by a read barrier that converted from-space pointers into to-space pointers by copying or following forwarding pointers as necessary. Brooks's goal was to reduce the cost of Baker's read barrier, moving most of the cost to generally less frequent writes. The technique is to relax Baker's to-space invariant, allowing mutator threads to observe objects in from-space if they have not yet been copied in the collection. To ensure that to-space copies are accessed if an object has been forwarded, Brooks gives each object a permanent forwarding pointer slot (which may be considerable space overhead if most objects are small). Pointer access operations *always* follow this forwarding pointer. When the object is a from-space object that has been copied to to-space, it will point to the to-space object; otherwise, it will point to itself. Figure ?? illustrates this organization. So the cost of a read is increased only by the cost of a second pointer read, rather than the cost of copying an object during GC. The collection must eventually reach a state where only to-space objects are reachable from the roots, so the scanning operation must eventually decrease the number of from-space pointers in to-space objects to zero. This progress property could be invalidated if mutators were allowed to write from-space pointers into already-scanned to-space objects. Brooks uses a write barrier to prevent such writes, changing the from-space pointer to a to-space pointer by copying if necessary before the write.

While the Brooks's algorithm certainly improves the worst case execution time analysis of programs compared to Baker's, pointer writes still incur significant worst-case cost. Henriksson's modification decreases this cost further, moving the difference to the collector task. As in the Brooks

collector, when the mutator writes a pointer field, it checks whether the new value points into from-space. If so, it allocates a to-space location for the from-space object, and alters the pointer value to point to the new to-space location. However, it does not yet copy the contents of the object. Instead, it inserts a "reverse" forwarding pointer into the to-space copy, pointing back to the from-space copy. (In figure ??, the to-space object's forwarding pointer would point back to its from-space version.) The standard Brooks trick of following the forwarding pointer on all accesses redirects accesses through pointers to the to-space copy back to the from-space copy. This state persists until the garbage collector (during its scheduled time) scans the object. It copies the object's contents from from-space, and resets the to-space copy's forwarding pointer to point to itself. Since allocation in a contiguous free space is fairly inexpensive compared to data copying, this clever technique effectively moves a good deal of the write barrier cost from mutator time into GC time, making worst-case execution times closer to a system without GC.

Bacon *et al.* use a rather different algorithm to achieve many of the same goals. They use an incremental mark-sweep algorithm with partial on-demand compaction. One motivation for this approach is the hope of requiring less space overhead for collection: the two-space copying approach of Baker (and Brooks, and Cheng and Belloch) requires the heap to be at least twice as large as maximum amount of live data, plus some extra amount to support allocation during incremental collection. The mark-sweep approach can typically support higher heap occupancies. Their marking algorithm uses the *snapshot-at-the-beginning* [?] approach. The collector attempts to determine reachability in a logical snapshot of the object graph that existed at the start of marking. Objects allocated during the marking are considered live; objects that existed before the marking, but were not marked, are reclaimed as garbage. Since the pointer graph may be modified by mutator updates to object pointer fields, such updates perform a write barrier before the field write. In this case, the write barrier examines the pre-write value of the field. If the referent is unmarked, it is marked and recorded for later processing traversal by the collector. As in Henriksson's algorithm, they don't completely escape the need to burden mutator operations with some extra overhead, but try to carefully engineer this overhead to make it as small as possible.

After sweeping to reclaim the garbage, compaction is done, if necessary, to ensure that enough contiguous free space exists to satisfy allocation requests until the next collection cycle. Here they use a technique very similar to Henriksson (though novel in the setting a mostly non-moving collector). They, too, use Brooks's permanent forwarding-pointer technique, requiring mutator accesses to object field to follow the forwarding pointer to find the object's current

location.

Bacon *et al.* claim that compiler optimizations can reduce the overhead of this technique by treating forwarding pointer reads as common subexpressions. This claim raises a subtle point. If a read of an object field translates to multiple machine instructions, can the mutator thread performing the read be preempted in the middle of those instructions? With collectors using the Brooks technique, the answer is no: the collector might run while the thread is preempted, and update the forwarding pointer to point to a to-space location. If the mutator resumes and completes the write to the from-space version of the the object, then the write will be missed. So the “Brooks barrier” must be atomic with respect to preemption. This by itself is not a great concern, since it is generally only a pair of instructions. But the optimization described by Bacon *et al.* suggests that one forwarding pointer read might then be used in a number of subsequent accesses to an object. The forwarding pointer read and all accesses that use it must collectively be atomic with respect to preemption. This works naturally in the cooperatively scheduled Jikes RVM [?] system in which they did their work, and they are careful to treat preemption points as killing forwarding pointer values.[?] But it should be noted that there is a tradeoff between the efficiency gained and extra jitter introduced by increasing the distance between legal preemption points.

There is perhaps a more important point with respect to this kind of optimization. While increasing average-case performance is useful, optimizations must apply to worst-case performance to really be relevant to hard real-time systems. So understanding of exactly when the optimization applies would have to be integrated into the timing analysis, which seems difficult.

Bacon *et al.* require copying of individual objects to be atomic with respect to mutator operations (so that each mutator operation on the object references either the old or new copy). This atomicity requirement prevents the collection thread from being arbitrarily preemptible, increasing its potential contribution to jitter.[?] To bound this jitter they bound the size of the largest individual copied chunk. Since arrays in particular may be large, they divide large arrays into fixed-sized chunks, and add a higher-level array containing pointers to the chunks. Here too they claim that compiler optimizations can eliminate the indirection overhead in many cases, but again the worst-case analysis would require integrated understanding of this optimization for it to be relevant to hard real-time systems. They do not discuss the issue, but a similar scheme could be applied to unusually large non-array objects.

Henriksson’s algorithm does not have this atomicity requirement: object copy operations are not considered complete until the to-space object’s forwarding pointer is updated to point to itself. Thus, copy operations can be aborted

at a fine grain, and simply restarted when the GC is next scheduled. While decreasing jitter, this approach has its own difficulties. There is a progress concern: if GC is repeatedly scheduled for very short periods, it may never succeed in copying a large object. Even if progress is usually made, calculations of the rate at which GC makes progress are affected, since each GC time slice must be considered to be shortened by the maximum amount of potentially aborted work. Therefore, techniques to bound the maximum copying increment are also relevant in Henriksson’s approach.

Table ?? summarize the characteristics of collectors we have discussed, including two examples each of work-based and time-based approaches. We give the costs each attaches to the mutator operations read, write, allocation, and (procedure call) return. We also give the effect on space overhead: the Metronome mark-sweep-compact collector is likely to have less space overhead than the semispace approach of the other three. For the time-based collectors, we list the operations that might effect the jitter – the longest atomic operation the collector might undertake while it is scheduled.

4.1. Garbage-First Collection

My own research group has implemented a collector that, while it is explicitly a soft rather than hard real-time collector, has some aspects in common with the time-based collectors described above. In the *Garbage-First* collector [?], the user inputs a real-time MMU specification, which we treat as a primary constraint: stop-world GC actions shall consume no more than x ms in any y ms window. Like the Metronome collector, we use snapshot-at-the-beginning concurrent marking (though we use true concurrency on multiprocessors), and interrupt the program periodically to perform compaction, reclaiming memory as a side effect. We divide the heap into equal-sized independently collectible *heap regions*, in a manner reminiscent of the Mature Object Space collector of Hudson and Moss [?]. In contrast to the fine-grained preemptibility of Henriksson and Metronome systems, in Garbage-First we assume that we can claim the machine periodically for up to the pause time permitted by the MMU specification. In such a *collection pause* we choose a *collection set* of heap regions from which we evacuate all live objects, reclaiming the collection set regions.

The Henriksson and Metronome systems complete (or abort) whatever fine-grained atomic GC operation is in progress when a GC scheduling activation is complete. This fine-grained preemptibility comes at the space and time cost of Brooks’s forwarding pointer. In contrast, Garbage-First avoids these overhead costs, but must be allowed to complete coarse-grained collection pauses once it starts them – these are our atomic GC operations. To meet the real-time constraint, we must not start collection pauses that will not

Work-based		
Baker [?]		
	read	copy 1 object
	write	none
	allocate	max(k units of collection work, flip = scan all stack threads)
	return	none
	space	semispace
Cheng & Blelloch [?]		
	read	none
	write	to both from- and to-space replicas
	allocate	max(k units of collection work, flip = scan 1 stacklet per thread)
	return	scan 1 stacklet
	space	semispace
Time-based		
Henriksson [?]		
	read	follow forwarding pointer
	write	reserve to-space for object, insert forwarding pointer
	allocate	max(constant allocate cost, constant flip cost)
	return	none
	space	semispace
	jitter	max(flip cost = scan all stacks, copy an object)
Bacon, Cheng & Rajan (Metronome) [?]		
	read	follow forwarding pointer
	write	marking write barrier
	allocate	constant
	return	none
	space	mark-sweep (better)
	jitter	max(flip cost = scan all stacks, copy an object)

Table ??. A comparison of example work- and time-based collectors.

be finished by the deadline. We achieve this by maintaining an internal model of the cost of collecting individual heap regions. This model has terms involving the size of the region's *remembered set* (a data structure that indicates heap locations outside the region that may contain pointers to objects in the region), and the amount of live data in the region. The concurrent marking process also counts the live data in each region, giving us an upper bound on the amount of data that might be copied and scanned in a collection pause. (At least for regions filled before the most recent marking started. If more recently allocated regions are included, their live data can only be estimated.) Given accurate estimates of the cost of collecting individual regions, and on fixed costs of collection pauses, we choose a collection set that can be collected in the available time.³ We choose as many of the most “desirable” regions, in terms of efficiency of collection, as will fit in the available time.

³ That is, “with high probability,” since some parts of the cost model are approximations – this is why Garbage-First is only a soft real-time collector.

This region ordering, which tends to pick the regions with the most garbage first, is what gives the collector its name.

To meet the real-time constraint, we must also ensure that collections do not occur too frequently. We track the start times and durations of all collections that might participate in a violation of the constraint in a data structure we call the *MMU tracker*. The MMU tracker allows us to determine the longest collection pause permissible at any point in execution, and also the earliest time in the future at which a pause of a given duration may start. Thus, if other heuristics would ordinarily initiate a collection pause, but a pause would violate the real-time constraint, the MMU tracker tells us how long we must delay that collection pause. Thus, we convert pause time issues into space issues.

5. Program Analyses for Real-Time GC

Hard real-time systems are often also *safety-critical*: a system failure is not just an annoyance, like some corporation losing some money, but a disaster, like someone losing their life. (I will leave the question of whether this is a dif-

ference of *degree* or of *kind* to the more philosophically inclined.) So a hard real-time system must meet its deadlines, no matter what. If a deadline is missed, the distinction between failure of worst-case timing analysis and running out of memory is probably of little interest. Therefore, if memory management in real-time systems is to be subjected to the same level of rigor as the rest of the system, we must be able to prove that the system doesn't run out of memory. This requirement holds equally for garbage-collected and explicitly-managed systems.

Baker, Henriksson, and Bacon *et al.* all present analyses for determining the heap size required to support a given application. These analyses relate the heap size to several properties of the application:

- the application's *maximum live data*: the maximum amount of data that is instantaneously reachable at any point in execution, and
- the application's *maximum allocation rate*: the amount of space allocated per unit time. This is usually the maximum over some given timing granularity related to the length of a GC cycle.

Baker's work-based algorithm is less interested in the allocation rate, since it attaches k units of GC work to allocations, automatically relating GC work rate to allocation rate. Therefore, his analysis of necessary heap size is parameterized by k rather than the allocation rate. Henriksson, and Bacon *et al.* both compute a minimum sufficient heap size from these inputs and properties of their algorithms.

Such analyses do not solve the end-to-end problem, however. How do we determine the maximum live data and allocation rates? "By measurement and testing" is an unsatisfying answer: this wouldn't suffice for timing analysis, so neither should it suffice for allocation and space usage analysis. Programs, even garbage-collected programs, sometimes have "storage leak" bugs, where their live data grows slowly, but without bound, over time. In safety-critical systems, we need static analyses that rule out such errors.

Determining the maximum allocation rate is probably the simpler of these two problems. Worst-case execution time analysis already determines the maximum execution time that can be consumed over all possible paths in the event handlers of a real-time system. It should be a fairly straightforward extension to instead (or in addition) determine the maximum amount of data allocated over any possible path, though I know of no work in this area. Then we would know the maximum amount of data that a thread could allocate in servicing an event. This can be combined with the real-time schedule to determine upper bounds on the maximum storage allocation over arbitrary periods. If a task T servicing a periodic event with period t has maximum allocation A when servicing the event, and we are interested in the system's maximum allocation over some pe-

riod t' , we must assume a contribution of $A(\lfloor t/t' \rfloor + 2)$ from task T . This is because an interval of length t' includes $\lfloor t/t' \rfloor$ full interval of length t , and two partial periods. T might have two different execution paths, each of which allocates A bytes, one doing all allocation at the beginning of the handler, and the other doing all the allocation at the end. Thus, we must include the full contribution from both partial intervals.

Bounding the maximum live data seems to be a harder problem. First, we should recognize that proofs of allocation and live data behavior are proofs not only about the properties of programs, but also involve assumptions on the acceptable ranges of "inputs." In determining allocation bounds, we may assume that events don't arrive more frequently than their specification allows. For maximum live data, the problem is more complicated, since it involves assumptions about the semantics of inputs, not just their inter-arrival times. Flatau [?] was the first (to my knowledge), to make some attempt to construct proofs of maximum live data usage, using the NQTHM theorem prover [?]. This work transformed the program into a new function (over the same inputs) that computed the maximum live data needed by the original program when computing those inputs, essentially by constructing a "slice" of the original program that abstracts away those portions of the computation that are not relevant to the amount of storage used. Flatau speculated that for many programs it would be possible to prove that simpler (but presumably less "tight") functions bound the space requirements. This work was done with safety-critical, but not necessary real-time, systems in mind.

More recently, Persson [?] made an innovative attempt to derive such bounds for real-time programs, and for general-purpose languages. This work has two basic ideas: first, types used in recursive data structures, such as linked list nodes, are annotated with bounds constraining the maximum depth of such recursion. This is somewhat familiar in this arena, since similar annotations are used to bound recursion and loops in worst case execution time analysis. Second, fields of recursive data structures are annotated to determine their effect on the analysis. For example, a linked list "next" field is distinguished as a *link* field, which points to a linked list whose size bound is one less than the parent's. Other fields can be labelled as *redundant* with some other pointer, and therefore do not contribute to the space bound. Persson uses the example of the "prev" pointer in a doubly-linked list. Without this annotation, a doubly-linked list node and a binary tree node are equivalent: both contain two recursive pointers to the containing type. However, space bounds resulting from a given recursion depth bound are very different for the two cases: $O(n)$ vs. $O(2^n)$! So these annotations are very important. This type analysis is extended to methods: each method is a source of roots, and therefore makes a contribution to the worst-case live data

size equal to the worst case sizes of data structures its local reference variables might reference, plus the maximum worst case size of any call it makes. The worst-case space usage is then just the worst space usage of the outermost method.

I view this work as a good start, but only that. The redundancy annotations require verification, which is not discussed. The *role analysis* of Rinard and Kuncak [?] might be useful here: this allows somewhat similar annotations, but requires specification of what other non-redundant reference a reference is redundant *with*. Persson does not discuss enforcement of recursion depth bounds. If these are enforced via some form of runtime check, perhaps via an exception, then this exception must be handled appropriately. (Though, again, this problem is similar to recursion and loop iteration limits, so existing techniques can be reapplied). Finally, the upper bounds reached via this analysis still seem like they may be made quite conservative by relatively common programming idioms. Redundancy specifications are only mentioned in the context of recursive data structures. The paper presents a `traverseList(1)` method that introduces an iteration variable `e` used to represent the current node in the loop. I read Persson's description of the technique as counting the same list at least twice in the worst-case bound: once for the `1`, and once for `e`. (And possibly more times, if the argument expression in the call to `traverseList` was itself a local variable.) It seems that method arguments should always be redundant, and that it should be possible to declare iteration variables such as `e` redundant (and verify, and perhaps infer, such assertions). So while there is considerable work left to do here, I find this an attractive conceptual framework from which to go forward.

6. Conclusions

In this paper I have tried to survey the history and current state of real-time garbage collection. I see a growing consensus that *time-based* methods, in which the garbage collector is treated as a separate task in the real-time scheduling problem, allow more realistic timing analysis of the real-time code, by decreasing GC "taxes" on mutator operations. Real-time garbage collectors rely on upper bounds on different aspects of application behavior, namely allocation rates and live data size. The most important remaining problems I see are not problems with collection algorithms, but rather with program analysis. Considerable rigor is applied to static analyses for determining worst-case timings. We need to apply the same level of rigor to worst-case timing analyses to allocation sizes and data rates.

7. Trademarks

Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

8. Acknowledgements

Thanks to the referees and my colleague Miriam Kadanisky for their comments. I would like to especially thank David Bacon for his careful reading and an interesting exchange of views.

References

- [1] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), Feb. 2000.
- [2] D. Bacon. private communication.
- [3] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92. ACM Press, 2003.
- [4] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, Jan. 2003. ACM Press.
- [5] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.
- [6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.
- [7] G. Bollella. *The real-time specification for Java*. Java series. Addison-Wesley, Reading, MA, USA, 2000.
- [8] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
- [9] R. A. Brooks. 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*, pages 256–262. ACM, ACM, Aug. 1984.
- [10] P. Cheng and G. Blleloch. A parallel, real-time garbage collector. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 125–136, N.Y., June 20–22 2001. ACM Press.
- [11] D. L. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. Technical report, Sun Microsystems Laboratories, 2004. To appear.

- [12] R. Fenichel and J. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.
- [13] A. D. Flatau. *A verified implementation of an applicative language with dynamic storage allocation*. PhD thesis, University of Texas at Austin, 1992.
- [14] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.
- [15] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, Lecture Notes in Computer Science, pages 388–403, St. Malo, France, Sept. 1992. Springer-Verlag.
- [16] V. Kuncak, P. Lam, and M. Rinard. Role analysis. *ACM SIGPLAN Notices*, 37(1):17–32, Jan. 2002.
- [17] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In K. Koskimies, editor, *Compiler Construction (CC'98)*, pages 80–94, Lisbon, 1998. Springer LNCS 1383.
- [18] J. O'Toole and S. Nettles. Concurrent replicating garbage collection. In *Conference on Lisp and Functional programming*. ACM Press, June 1994.
- [19] P. Persson. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, volume 34.7 of *ACM Sigplan Notices*, pages 45–54, NY, May 5 1999. ACM Press.
- [20] S. G. Robertz and R. Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.
- [21] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.