Syncopation: Generational Real-time Garbage Collection in the Metronome

David F. Bacon
IBM Research

Perry Cheng
IBM Research

David Grove
IBM Research

Martin T. Vechev
Cambridge University

Abstract

Real-time garbage collection has been shown to be feasible, but for programs with high allocation rates, the utilization achievable is not sufficient for some systems.

Since a high allocation rate is often correlated with a more high-level, abstract programming style, the ability to provide good real-time performance for such programs will help continue to raise the level of abstraction at which real-time systems can be programmed.

We have developed techniques that allow generational collection to be used despite the problems caused by variance in program behavior over the short time scales in which a nursery can be collected. *Syncopation* allows such behavior to be detected by the scheduler in time for allocation to by-pass the nursery and allow real-time bounds to be met.

We have provided an analysis of the costs of both generational and non-generational techniques, which allow the trade-offs to be evaluated quantitatively. We have also provided measurements of application behavior which show that while syncopation is necessary, the need for it is rare enough that generational collection can provide major improvements in real-time utilization. An additional technique, *arraylet pre-tenuring*, often significantly improves generational behavior.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.3.2 [*Programming Languages*]: Java; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); D.4.7 [*Operating Systems*]: Organization and Design—Real-time systems and embedded systems

General Terms Experimentation, Languages, Measurement, Performance

Keywords Scheduling, Allocation, Real-time, Garbage Collection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.

Copyright © 2005 ACM 1-59593-018-3/05/0006. . . $5.00.

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 [9]) 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 [6, 5] we showed that it is possible to build a provably real-time garbage collector for a language like Java. A key aspect of this work was the use of *time-based* scheduling instead of the work-based scheduling approach that had been in common use since Baker's original work on real-time collection [7]. Our collector is called the *Metronome* because it alternates between the mutator and the collector with extremely regular "ticks".

The result was a collector that is able to guarantee a minimum mutator utilization of 50% at a resolution of 10ms: out of every 10ms, the mutator threads receive no less than 5ms — with no exceptions. During periods when collection is off, the mutators receive almost all of the CPU (a small portion is charged to the collector for things like allocation operations). Collection is active about 45% of the time, resulting in good but not exceptional throughput.

There was significant interest from potential users of this technology. As we met with them we found that there were two major barriers to the adoption of our system, one technical and one practical. The technical problem was that a utilization level of 50% during collection was not acceptable to some users: they wanted more CPU time available for real-time tasks even while garbage collection was active.

Note that this is *not* a throughput issue, but rather a utilization issue: a constant 20% reduction in utilization was acceptable, while oscillation between 0% and 50% reduction is not.

Unfortunately, it is undesirable to simply spread out the collection more evenly, since this would allow the mutator to allocate more memory while collection is in progress. The increased allocation in turn could cause the memory budget to be exceeded, leading to a failure to meet real-time bounds.

The second barrier to the adoption of the system was practical: they required a complete Java development environment with full library and debugging support. The Metronome was implemented in Jikes RVM [1], which lacks these attributes.

Therefore, we set out to build a collector with the same predictability as the Metronome, but with higher utilization and in-

creased throughput — in a production quality JVM, IBM's J9 virtual machine [16].

To achieve higher utilization, we use generational collection [23], which focuses collector effort on the portion of the heap most likely to yield free memory. This has added benefits in a concurrent collector because it reduces the amount of floating garbage, which is typically a drawback of such systems. However, collecting the nursery is unpredictable both in terms of the time to collect it and the quantity of data that is tenured.

Real-time systems are fundamentally different in that heuristic optimizations that are not monotonic can not be applied. Standard generational collection is not always better; it is merely often better. Furthermore, standard generational collection has no fixed bounds on the time required to collect the nursery. As a result, there are considerable additional complexities that are involved in applying generational techniques to a real-time collector.

The contributions of this work are:

- An algorithm for generational real-time garbage collection based on synchronous nursery collection, which can significantly increase throughput and reduce memory consumption;
- An analytic solution for the achievable utilization of both generational and non-generational collection, allowing the correct collector to be selected for a given set of application and virtual machine parameters.
- *Syncopation*, a technique for handling temporary spikes in allocation rate that would otherwise make it impossible to evacuate the nursery within the real-time bounds;
- *Arraylet pre-tenuring*, a technique that significantly increases the effective nursery size without increasing the cost of evacuation, thereby increasing utilization and reducing floating garbage; and
- Measurements of applications showing the potential effectiveness of both syncopation and arraylet pre-tenuring.

The paper is organized as follows: Section 2 provides background on the Metronome real-time garbage collector. Section 3 describes our basic approach to generational collection and develops analytic results for its efficacy. Section 4 introduces syncopation and Section 5 describes arraylet pre-tenuring. We then discuss related work in Section 6 and draw our conclusions.

2. Metronome Overview

We begin by summarizing the results of our previous work [6, 5] 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. Section 3 will describe our generational extensions and modifications.

The Metronome is a hard real-time incremental uni-processor collector. It uses a hybrid of non-copying mark-sweep collection (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 deterministic, which is a crucial property for real-time collection.

The Metronome implementation described in previous work was done in the Jikes RVM Java virtual machine from IBM Research [1]. The new system described in this paper is being built in J9, one of IBM's production virtual machines. For typical applications, the J9 implementation is real-time at 10 milliseconds with 70% minimum mutator utilization (MMU). In other words, the collector *without exception* used less than 3 milliseconds in any given 10 millisecond window.

The key elements of the design and implementation of the Metronome collector are:

Time-based Scheduling. The Metronome collector achieves good minimum mutator utilization, or MMU, at high frequencies (1024 Hz) because it uses time-based rather than work-based scheduling. Time-based scheduling simply interleaves the collector and the mutator on a fixed schedule.

Guaranteed Real-time Bounds. Despite our use of time- rather than work-based scheduling, we are able to tightly bound memory utilization while still guaranteeing good MMU.

Incremental Mark-Sweep. Collection is a standard snapshot-at-the-beginning incremental mark-sweep algorithm [24] implemented with a weak tricolor invariant [18]. 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.

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. If a page becomes fragmented due to garbage collection, its objects are moved to another (mostly full) page containing objects of the same size [8].

Read Barrier. Relocation of objects is achieved by using a forwarding pointer located in the header of each object [10]. A read barrier maintains a to-space invariant (mutators always see objects in the to-space).

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.

Metronome only runs on uniprocessors. This choice was made because virtually all embedded systems are uniprocessors and the resulting simplification allows much more efficient implementation. In particular, 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 scheduling quantum in which the collector runs.

The Metronome uses time-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [7], has used work-based scheduling. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The problem with Baker's definition is that it defines real-time behavior in terms of the collector, rather than the application. Real-time tasks require a guarantee that they can execute to completion in a given (short) interval. Baker's work-based methodology merely provides bounds on individual collector operations, but is unable to bound the CPU consumption of the collector within a given interval. Therefore, completion guarantees cannot be given to the tasks.

This becomes clear when viewed from the perspective of real-time systems methodology. Work-based collectors represent aperiodic event-triggered tasks of varying cost. The combined non-determinism of the events and of their costs results in a worst-case execution time (WCET) that can (and often does) consume the entire real-time period, leaving the application with an MMU of 0.

Previous collectors avoided time-based scheduling out of fear that during periods of heavy allocation, the collector would fall behind and be forced to stop the mutator in order to complete.

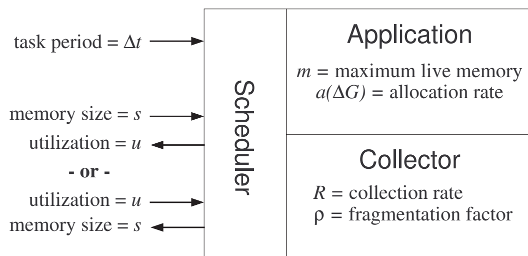


Figure 1. Interaction of components in a Metronomic virtual machine. Parameters of the application and collector are intrinsic; parameters to the scheduler are user-selected, and are mutually determinant.

The Metronome 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 $a(\Delta G)$. 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, and the resolution at which the utilization is calculated Δt.

The relationship between these parameters is shown graphically in Figure 1. The mutator is characterized by its allocation rate over the interval of a garbage collection $a(\Delta G)$ and by its maximum memory requirement m. The collector is characterized by its collection rate R. The tunable parameters are Δt, which controls the frequency of collector scheduling, and either the CPU utilization level of the application u thus determining memory size s, or a memory size s which determines the utilization level u.

3. Generational Collection

In some cases the Metronome may not be able to meet an application's real-time utilization requirements. In that case, there are a number of things the programmers can do: they can increase space consumption s by buying more memory; they can decrease the utilization requirement u by buying a faster processor; they can reduce the allocation rate a by rewriting the code to perform less dynamic allocation; or they can reduce the live memory size m by rewriting the code to reduce the size of long-lived data structures.

However, to the greatest possible extent we wish to avoid placing such a burden on the user. Within the system, there are several ways to improve matters: speed up the collection rate R by tuning the collector; decrease m by using various compaction techniques; or decrease the fragmentation factor ρ by improving the heap architecture and free space management.

Unfortunately none of these techniques is likely to provide the order-of-magnitude improvements that we require. The original Metronome collector had already been fairly well tuned; fragmentation was bounded fairly tightly (12% in theory and 3% in practice); and we have already applied object model compression techniques to the J9 virtual machine [4] in which we are implementing the collector described in this paper.

However, there is a way in which we can reduce the allocation rate a. If we employ generational techniques, we can view the nursery as a filter which reduces the allocation rate into the primary heap to $a\eta < a$, as shown in Figure 2. In general we expect $\eta \ll 1$, which will greatly reduce the load on the main collector. Of course, collecting the nursery will also have a cost, but for most

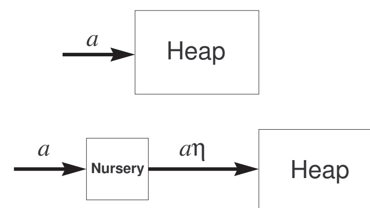


Figure 2. For a program with allocation rate a, interposing a nursery reduces the effective allocation rate to $a\eta$, where η is the nursery survival rate.

applications we expect that the benefits of reducing the workload on the main collector will normally outweigh these costs by a large margin.

3.1 Synchronous Nursery Collection

Typical stop-the-world generational collectors consist of two disjoint collectors: one for the nursery and the other for the tenured (heap) space. Both collectors are usually run with mutators stopped. In a scheme where heap collection is incremental, care must be taken to synchronize with nursery reclamation. Moreover, the incremental collector requires the use of a snapshot write barrier. Therefore, the write barrier must provide both generational and snapshot functionality.

Nursery collection is typically triggered when the nursery is full. In an incremental system mutators can interleave with the collector. Therefore a mutator can evacuate the nursery either when the collector is not running or when the collector is marking or sweeping. In addition, the incremental collector itself evacuates the nursery at the beginning of its root scanning phase. There are three optimizations arising from this step. First, the snapshot write barrier does not need to record the overwriting of nursery to heap pointers. Secondly, during its heap marking phase, the incremental collector does not need to trace nursery objects. Thirdly, we can make sure that we eliminate all of the floating garbage in the nursery. If we choose not to evacuate in the beginning of our collection cycle, then the above optimizations cannot be applied. The incremental collector also performs nursery evacuation at the start of its sweeping phase.

Aside from when nursery evacuation occurs, another effect of combining generational and incremental collectors is the write barrier operation. Although both write barriers protect against the loss of a reachable object, the snapshot and the generational barrier share the following fundamental differences:

- Generational barriers are *always* active, snapshot barriers are *partially* active: only when the collector is heap marking.
- Generational barrier entail object/region rescanning for pointer fixup, snapshot barriers do not.
- Generational barrier remembers the destination object/region, snapshot barriers remember the overwritten pointer.
- Generational barrier performs range comparisons to determine whether the new pointer is a heap to nursery one, the snapshot barrier does not perform range checks.

Most pointer stores are nursery to nursery pointers. Since nursery collection is synchronous, we do not require a snapshot write barrier on those pointers. Additionally, since we evacuate the nursery at the start of root set scanning, we also do not need to barrier nursery to heap pointers. Therefore, the common write barrier can now filter on the destination object. That is, if the destination object is in the nursery, we do not need either of the two barriers. For

snapshot, we also do not need to barrier heap to nursery pointers, which are needed by generational barrier. Conversely, the snapshot barrier needs to record heap to heap pointers while the generational barrier does not.

Therefore, with this new insight, we can utilize the range checks that the generational write barrier performs on the destination object in order to to filter out significant number of snapshot barriers. We note how write barrier operation is connected with the timing of the nursery evacuation.

In a real-time environment, if we are performing synchronous nursery collection then we must be able to compute the worst-case execution time (WCET) for nursery collection. This means carefully bounding all possible sources of work. In particular, the remembered set is also allocated from within the nursery. Objects are allocated left to right, and remembered set entries from right to left. When the two regions meet, the nursery is full and must be collected.

3.2 Utility of Generational Collection

To begin with we require a method for determining whether generational collection will provide a benefit. Surprisingly, we found no analytical model for this in the literature. Analytical models which rely on naive assumptions about application characteristics naturally have limited use, but on the other hand the demands of real-time systems require more stringent analysis of performance effects. Since the aim of generational collection is to improve utilization in the real-time interval, it is of little use if generational collection "usually" or even "almost always" improves performance.

Therefore, we have developed an analytical model which we present here. The garbage collector itself is characterized by the following parameters:

- N is the nursery size (bytes);
- R_T is the tracing rate in the heap (bytes/second);
- R_S is the sweeping rate in the heap (bytes/second);
- R_N is the collection rate in the nursery (bytes/second);

The application is characterized by the following parameters:

- a is the allocation rate (bytes/second) assuming infinitely fast garbage collection;
- m is the maximum live memory of the mutator (bytes);
- η is the survival ratio in the nursery. (Dependent on N.)

3.3 Time and Space Bounds

We characterize the real-time behavior of the system with the following parameters:

- Δt is the task period (seconds);
- u is the minimum mutator utilization in each Δt;

From the above parameters, we can then derive the overall space consumption of the system:

- s is the space requirement (bytes) of the application in our collector.

The allocation rate a and the survival ratio η in fact vary considerably as the application runs. For the time being we will consider the case when they are smooth. In Section 4 we will describe the effect of variations in those parameters, and how the implementation copes with the variations.

For a given interval Δt, the collector may consume up to $(1-u)\cdot\Delta t$ seconds for collection. We define the *garbage collection factor* γ as the ratio of mutator execution to useful collector work.

$$\gamma = \frac{u\cdot\Delta t}{(1-u)\cdot\Delta t} = \frac{u}{1-u} \quad (1)$$

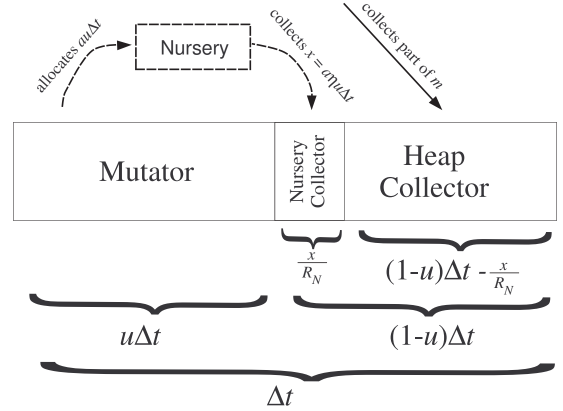


Figure 3. Time dilation due to generational collection causes additional allocation during a major heap collection, but attenuates all allocation by the survival rate η.

Multiplying by γ converts collector time into mutator time; dividing does the reverse. Since the relationship between u in the range $[0,1)$ and γ in the range $[0,\infty)$ is one-to-one, we also have

$$u = \frac{\gamma}{1+\gamma} \quad (2)$$

To bound the space required by the collector in order to maintain real-time bounds, we need to know how much extra space may be allocated during a collection cycle. In the absence of generational collection, the extra space e_M for the Metronome is

$$e_M = a\gamma\cdot\left(\frac{m}{R_T}+\frac{s}{R_S}\right) \quad (3)$$

which is the allocation rate multiplied by the time required to perform a collection.

When generational collection is introduced, the allocation is attenuated by the survival rate η. However, performing generational collection is not free, so it takes longer to collect the main heap. This in turn means that the mutator performs more allocation during collection. This effect is shown in Figure 3, expressed by the following equations, in which we define the generational *dilation factor* δ and the corresponding extra space e_G under generational collection:

$$\delta = 1-\frac{a\eta}{R_N}\cdot\gamma \quad (4)$$

$$e_G = \frac{a\eta\gamma}{\delta}\cdot\left(\frac{m}{R_T}+\frac{s}{R_S}\right) \quad (5)$$

Freeing an object in our collector may take as many as three collections: (1) the first is to collect the object; (2) the second is because the object may have become garbage immediately after a collection began, and will therefore not be discovered until the following collection cycle — floating garbage; and (3) the third is because we may need to relocate the object in order to make use of its space. The first two properties are universal; the third is specific to our approach.

As a result, the space requirement of our collector paired with a given application is

$$s = (m+3e)\cdot(1+\rho) \quad (6)$$

where e is e_M or e_G, and ρ is the fragmentation factor, which is a settable parameter (we typically use $\rho = 1/8$). Because the

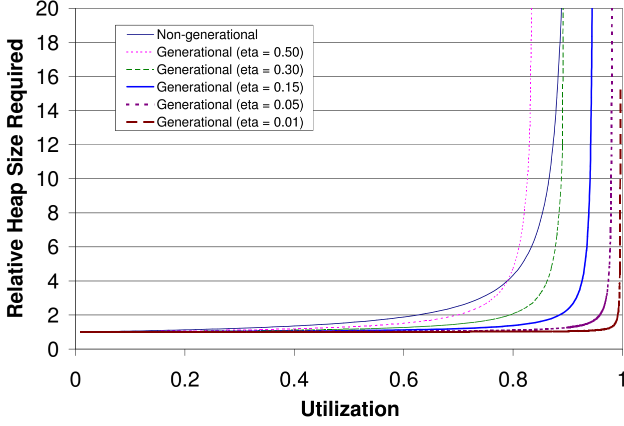


Figure 4. Relative Space Usage vs. Utilization. Low allocation rate: $a = 20$ MB/s.

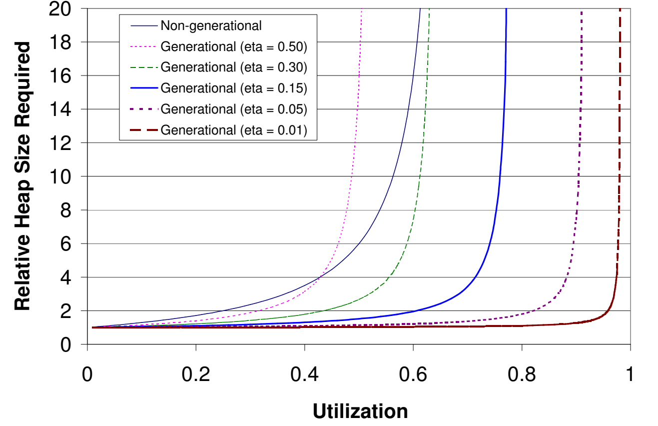


Figure 5. Relative Space Usage vs. Utilization. High allocation rate: $a = 100$ MB/s.

fragmentation factor is uniformly present and because the nursery size N is negligible compared to m, we can simply adopt the fragmentation-free heap-size multiplier formulation to measure space expressed by

$$\sigma = \frac{s}{m \cdot (1 + \rho)} \tag{7}$$

Substituting equations 3 and 5 into equation 7 gives us the relative space bounds for the original Metronome collector and the generational version.

$$\sigma_M = \frac{R_T R_S + 3a\gamma R_S}{R_T R_S - 3a\gamma R_T} \tag{8}$$

$$\sigma_G = \frac{\delta R_T R_S + 3a\gamma\eta R_S}{\delta R_T R_S - 3a\gamma\eta R_T} \tag{9}$$

These equations show what the space consumption will be for some utilization u since γ is a function of u given by equation 1. These equations can be inverted to give the achievable utilization in terms of a maximum heap size.

$$u_M = \frac{\sigma_M - 1}{3a\left(\frac{1}{R_T} + \frac{1}{R_S}\right) + \sigma_M - 1} \tag{10}$$

$$u_G = 1 - \frac{3a\eta\left(\frac{1}{R_S} + \frac{1}{R_T}\right)(1 + \frac{a\eta}{R_N})}{\sigma_G - 1} \tag{11}$$

It is because of the factor of three multiplier that generational collection is so valuable — much more so than in a stop-the-world generational collector. As long as the survival rate η is even moderately low, the space required to meet real-time bounds is reduced almost by a factor of $3/\eta$.

It should be noted that the expected space requirement is on the order of $m + e$, because although the second and third additional collections described above could apply to all objects, in practice they only apply to a very small percentage.

3.4 When to Use Generational Metronome?

When the survival rate η is extremely low, the generational collector will be very effective because the work necessary to maintain available space is also very low. On the other hand, when the survival rate nears 1, the nursery provides no benefit while imposing an extra copy on every allocated object. Intuitively, there is a crossover point when the use of nursery neither helps nor hurts the Metronome. At this point, the space consumption and utilization would be jointly equal. Inspection of equations 8 and 9 reveals that this condition is met when $\eta = \delta$. A more rigorous derivation, which we omit for conciseness, reveals that there are no other solutions. This crossover condition can be re-formulated to express the cross-over utilization u_C given a fixed survival rate, or conversely to give the cross-over survival rate η_C.

$$u_C = \frac{R_N \cdot (1 - \eta)}{a\eta + R_N \cdot (1 - \eta)} \tag{12}$$

$$\eta_C = \frac{R_N}{R_N + a \cdot \frac{u}{1-u}} \tag{13}$$

Figures 4 and 5 show the relative heap size required as a function of a target utilization for a low ($a = 20$ MB/s) and high ($a = 100$ Mb/s) allocation rate (see equations 8 and 9). The curves compare the non-generational Metronome against the generational version at several different survival rates. The remaining parameters are fixed: the heap tracing rate $R_T = 150$ MB/s, the heap sweeping rate $R_S = 600$ MB/s, and and the nursery collection rate $R_N = 75$ MB/s.

Every curve has the same shape. At low utilizations, the space consumption is low (although for real systems, utilization levels under 50% are unlikely to be acceptable). As the utilization is increased, the space consumption at first increases slowly, reaches an inflection point, and then rises very rapidly. In a generational system, because the nursery collection and the concurrent collection compete for time, the knee in the curve is somewhat sharper than that in the non-generational system. Practically, this means that when using a generational system, care must be taken to avoid running too close to a regime where the space consumption might explode.

When the space consumption spikes, the system is unable to keep up with the allocation rate. In the non-generational version, the sweep rate is throttling the mutator's progress because memory is unavailable for allocation until the desired memory has been swept. In the generational version, the nursery tracing rate (mitigating the the survival rate) will throttle the program because the nursery is evacuated synchronously. In either case, when the system falls behind, the memory usage becomes unbounded.

An intuitive and graphically obvious trend in the curves for the generational system is that lower survival rates allow a higher achievable utilization level. On the other hand, when the survival rate is high, the nursery is so ineffective that it will lose to the non-generational version. For example, Figure 4 shows that when the survival rate is 50% and the allocation rate is low, the genera-

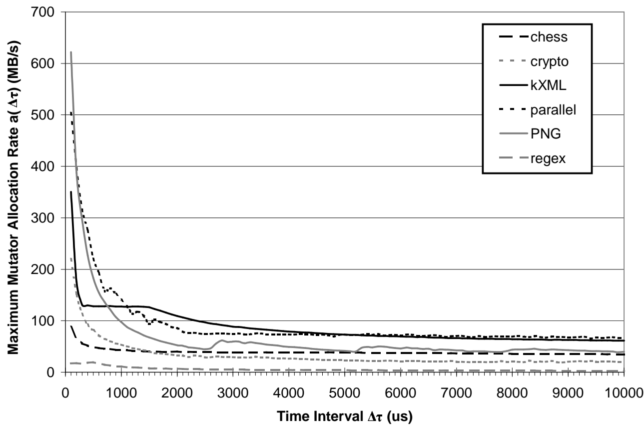


Figure 6. Maximum Mutator Allocation Rate (MMAR) for the EEMBC embedded benchmarks running on a 2.4 GHz Pentium processor. Allocation rates converge at intervals above 1-3 milliseconds.

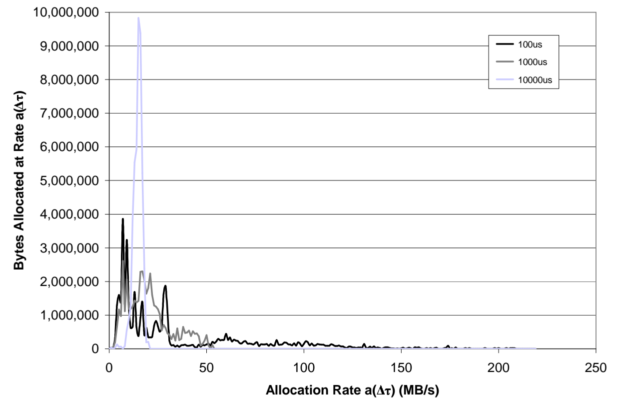


Figure 7. Need for Syncopation (EEMBC crypto benchmark). With a very short real-time interval $\Delta\tau = 100\mu s$, the application allocates as much as 250 MB/s in some intervals; at $\Delta\tau = 10000\mu s$, the peak allocation rate drops to 25 MB/s.

tional system becomes unusable around $u = 0.70$ while the non-generational system (Figure 5) is able to function at $u = 0.77$.

4. Syncopation

The analysis of the previous section contains two important (and unrealistic) simplifying assumptions, namely that the allocation rate a and the survival rate η are uniform in each time interval Δt. Since nursery collection is synchronous and the nursery size is tuned to Δt, if the allocation rate spikes temporarily, the nursery will fill more than once per interval Δt and the collector will fall behind the mutator and perform multiple synchronous nursery collections per interval, at which point the desired utilization u can no longer be guaranteed.

The Metronome is able to use time-based scheduling because it requires the application to specify a limit on the allocation rate *at the period of the collector*. If the real-time interval Δt is 1 millisecond and the collector period ΔG is 1 second, then the allocation rate can be averaged over a time interval 1000 times longer. Fundamentally, this allows us to bound the WCET of the collector over its own period, and then divide up that time evenly and schedule it as a time-triggered task.

However, by introducing synchronous collection of the nursery, it would appear that we are doomed to reduce the period over which we can average the allocation rate and survival rate from the collector period down to the real-time interval: while the major heap collector is a long-period task, nursery collection is a high-frequency task. Since it is performed synchronously, it will most likely be the limiting factor on our ability to drive down pause times.

The quality of service provided by a real-time garbage collector is measured by its *minimum mutator utilization* or MMU [11], which is the minimum amount of time provided to the mutator in a particular interval.

In order to evaluate the effect of allocation behavior on real-time collection, we define an analogous quantity: *maximum mutator allocation rate* or MMAR. For a given time interval $\Delta\tau$, the MMAR of a program is the highest allocation rate of any $\Delta\tau$-size interval of the program's execution. MMAR must be measured as though garbage collection takes zero time, since otherwise a poorly implemented collector would be subjected to a lower allocation rate.

The MMAR curves of the six EEMBC benchmark programs [15] are shown in Figure 6, for $100\mu s \leq \Delta\tau \leq 10ms$. Surprisingly, the allocation rate stabilizes very rapidly, between 1 and 3 milliseconds. This shows that if the nursery is able to absorb short-term allocation bursts on the order of a few milliseconds, a synchronous generational collector should be able to provide good utilization (MMU).

However, there is no guarantee that short-term bursts will subside by any particular point in time, and we also wish to drive the maximum pause times below 1 millisecond. However, as we shorten the time interval, we will begin to encounter intervals in which there is more allocation than can be collected synchronously.

This is shown for the `crypto` benchmark in Figure 7: for three different time intervals $\Delta\tau = 100$, 1000, and $10000\mu s$, we plot the allocation rate in the interval $\Delta\tau$ versus the number of bytes allocated at that rate. As the time scale increases from $100\mu s$ to $10ms$, the tail starts to disappear and the curve shifts to the left, allowing progressively slower collectors to keep up. At 10ms, the curve has almost degenerated into the single point corresponding to its overall average allocation rate.

There will be some allocation rate above which the nursery collector will not be able to keep up. The area under the curve to the right of that point is the amount of memory that is handled via *syncopation*.

In music, syncopation is the placement of emphasis on a usually unstressed beat. In our collector, *syncopation* is the movement of collection work from a stressed interval to an unstressed interval.

When the collector determines that the product of $a\eta$ is too high, it pre-tenures objects until the allocation rate subsides again.

4.1 Syncopation via Allocation Control

We present two alternative approaches to syncopation. In the first we dynamically resize the nursery (virtually, not physically) to ensure that we can still perform a worst-case evacuation of the nursery without violating the mutator utilization requirement. For example, if $\Delta t = 10ms$ and $u = 0.7$ and if the first nursery collection in the interval consumes 2 ms, then we can still consume 1 ms for collection. If we resize the nursery so that its collection has WCET=1 ms, then the system can proceed safely.

When the nursery becomes too small to be useful (because $\eta \rightarrow 1$), we change allocation policy and simply allocate all objects directly in the heap; we call this *floodgating*. During such a period, the effective allocation rate into the heap will spike from the nursery-attenuated rate $a\eta$ to the full rate a. The system continues to dynamically monitor the MMU, and when it has risen sufficiently it switches back to a nursery allocation policy.

The advantages of this approach are that it is effective and relatively simple to implement. The disadvantages are that it requires provisioning for the worst case in advance, which means that it can not make full use of available processing resources, and that it can only adapt to variations in a, but not in η. Furthermore, it requires a conditional on the critical path of the inlined allocation sequence, which slows down all allocations.

To understand why floodgating precludes the full use of an available collector quantum, consider the case without floodgating in which we consume a full collector quantum $(1-u)\Delta t$ to collect the nursery. It is very desirable to be able to do so since it means that we can use the largest possible nursery, which will maximally attenuate η.

However, when we finish collection we have used our full quantum, so we must guarantee that the mutator will run for at least $u\Delta t$ before any collection takes place. However, there is no bound on the instantaneous allocation rate of the mutator. It could fill the nursery within $u\Delta t/2$ time units, at which point we would be forced to synchronously evacuate the nursery and would fail to meet our MMU commitment.

The only alternative is to immediately begin tenuring newly allocated objects and not allow any nursery allocation for $u\Delta t$ time (or at least $u\Delta t - \epsilon$, where ϵ is a time interval so short that it is impossible for the mutator to fill the nursery). At that point, we perform another collection quantum, and are back to where we started: being unable to allocate into the nursery.

4.1.1 Scheduling: Multiple Beats per Measure

The syncopation techniques in and of themselves are fairly simple. The difficulty lies in scheduling them in such a way that MMU requirements are met and memory and processing resources are not squandered.

Because of the pessimistic property of allocation-based syncopation, some over-provisioning is necessary. This is done by increasing the frequency of collection operations within Δt. Continuing our musical analogy, a time interval Δt is called a *measure*, and the sub-divisions of a measure are called *beats*, of which there must be an integral number. Utilization must be expressed as a number of beats per measure. Taken together, these quantities comprise the *time signature* of the system.

Since the allocation rate could spike at any time, it is possible that two nursery collections could be forced to happen almost back-to-back. Therefore we retain one beat in reserve. Syncopation occurs when a second, consecutive collector beat consumes that reserve. At that point, allocation changes to immediately tenure all objects until a one-beat reserve has been reclaimed.

In this approach we can also dynamically adapt to changes in the survival ratio η, because after a nursery collection which does not consume a full beat, we can resize the nursery so that the WCET for its collection is bounded by our remaining fractional beat. Of course, this is not worthwhile below a certain fraction of a beat, since the nursery becomes so small that $\eta \rightarrow 1$, and we are simply implementing a more expensive method of immediate tenuring.

The fundamental limit of such an approach is that it requires that a nursery can be collected within a single beat, which is considerably smaller than the total collector quantum $(1-u)\Delta t$. Thus this approach is limited in its ability to drive down the fundamental time period of the collector. In practice, we find that for our particular hardware and benchmarks, it works well down to a measure length of about $\Delta t = 10ms$, with 10 or 20 beats per measure, requiring a WCET for nursery collection of $500 - 1000\mu s$.

4.2 Syncopation via Collection Control

The limitations of allocation control give rise to an alternative method of syncopation based on controlling the collection, rather than the allocation.

In this regime, the collector begins collecting the nursery at the beginning of the collector quantum. If it completes collection in time, it resets the nursery and resumes the mutator. However, if the collector quantum expires before nursery collection is complete, it syncopates: it unconditionally tenures the remaining unevacuated objects by logically moving the nursery into the mature space. This is done by appending it to a list of *off-beat* pages. Then a new nursery is obtained from a pre-allocated reserve and the remembered set buffer is reset. Since all these operations are done logically, by redirecting pointers, syncopation is extremely fast.

This method of syncopation is made possible by the presence of a read barrier in the collector. As the nursery is being collected and objects are moved into the heap, the forwarding pointer that is left behind has the exact same format as the forwarding pointer used to facilitate incremental object movement in the main heap. Therefore, when a partially collected nursery is tenured, heap to nursery references that were stored in the remembered set but not yet fixed will simply follow the forwarding pointer via the read barrier.

The major advantage of collection-based syncopation is that it allows the collector to consume a full mutator quantum, without requiring a change in the allocation policy. If the allocation rate subsides in the subsequent quantum, the collector will immediately regain the full benefits of generational collection.

The primary disadvantage is fragmentation: even if only one object in the nursery is live, none of the memory will be reclaimed until a full collection has taken place. Even worse, since the nursery was allocated sequentially rather than using segregated free lists, it must be completely evacuated before it can be reclaimed, increasing the defragmentation load and making the memory unavailable for one and a half major collection cycles (one to move the live objects, the other to forward any pointers to them).

In the limit, collection-based syncopation degenerates into a semi-space collector, in which all nurseries are syncopated and become the from-space. However, it would be almost impossible to cause this to happen, even with an adversary program.

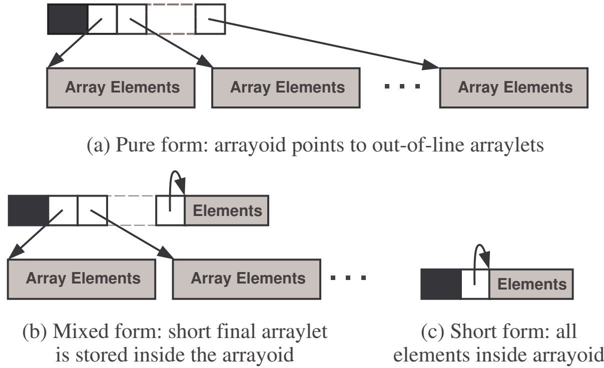
While syncopation makes it possible to use a full collector quantum for collection, and to allow the nursery to grow to a point beyond its WCET collection limit, it is undesirable to do so. Thus it may still be desirable to have multiple collector beats per measure, as in allocation-based syncopation, although we expect that significantly fewer beats will be required.

4.3 Cost of Syncopation

Syncopation is a technique that allows us to time-average a short-term cost variance by moving the work to a later point in time. Since we have a bound on the long-term allocation rate, we know that any rate spike will be accompanied by a corresponding rate drop within the collector period.

However, at a certain point the variance in the behavior of the mutator is so high that the cost of syncopation will outweigh its benefits. In particular, consider the case when all memory to be allocated in an entire major collection cycle ΔG is allocated within a single interval Δt. In that case, the beat generation will contain virtually all of the allocated objects, and we will have received none of the benefits of generational collection *even if the program obeys the generational hypothesis*.

Variation in the allocation rate and survival rate must be considered together because they drive the load on the nursery collector in concert.



(a) Pure form: arrayoid points to out-of-line arraylets

(b) Mixed form: short final arraylet is stored inside the arrayoid

(c) Short form: all elements inside arrayoid

Figure 8. Arraylets.

The work performed by nursery collection is linear in the size of the surviving data, so as long as the nursery collection rate R_N is higher than the survival rate $a\eta$, no syncopation is required. However, above that point the survival rate will only be attenuated by η the first time the nursery fills up; after that memory will be allocated into the heap at the unattenuated rate a.

For a particular time interval Δt_i with allocation rate a_i and survival rate η_i, the effective allocation rate into the heap a'_i is

$$a'_i = \begin{cases} a_i \eta_i & \text{when } a_i \eta_i \leq R_N, \\ a_i - \frac{R_N \left(\frac{1}{\eta_i} - 1 \right)}{\gamma} & \text{otherwise.} \end{cases} \quad (14)$$

While the extra space cost due to syncopations could be calculated precisely if all a_i and η_i were known, such a specification would be very cumbersome and lack abstraction for the user.

We are currently investigating ways for the user to bound the potential extra load on the collector by concisely describing the behavior of the application (in terms of the variation shown in Figure 7). In general, the trade-off imposed by generational collection is higher utilization in exchange for more specific information about program behavior.

5. Increasing Mortality

In the previous section we saw that the performance of the system is critically determined by the nursery survival rate η. In this section we describe a technique for decreasing the survival rate by increasing the effective size of the nursery.

In the Metronome, an array consists of two parts: an *arrayoid*, which contains the object header and pointers to *arraylets*, which are contiguous aligned chunks of size Σ. If the last arraylet is smaller than the arraylet chunk size minus the arrayoid header size, then it is allocated contiguously with the arrayoid itself. This leads to three basic array organizations, as shown in Figure 8.

The arraylet size Σ is determined by the page size Π and the desired fragmentation ratio ρ, such that $\Sigma = \rho\Pi$. Maximum immediately allocatable array length is thus limited to $\Pi\Sigma/4$ (assuming 4 bytes/word). In our implementation we use page size $\Pi = 16$KB and fragmentation ratio $\rho = 1/8$, for an arraylet size of $\Sigma = 2$KB and a maximum immediately allocatable array length of of 8MB. Larger arrays which may require arbitrarily large contiguous memory must be requested via a potentially blocking interface, which may have to wait up to 2 collection cycles for sufficient contiguous memory to be evacuated and its incoming pointers forwarded.

The mixed arraylet form of Figure 8(b) is required to maintain the fragmentation bound, since an array of size $\Sigma + 1$ allocated in the pure form (a) would otherwise consume two arraylets for a fragmentation of 50%. By using the mixed form we can bound the fragmentation by ρ.

Note that in order to be power-of-two aligned, the arraylets contain no internal metadata; this is kept in the per-page metadata, which must be consulted at page boundaries when parsing the heap.

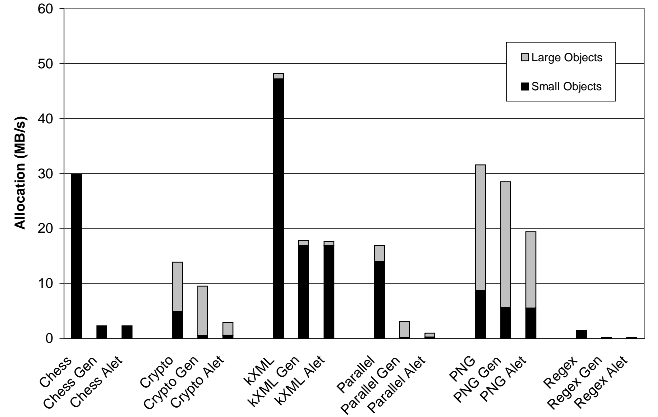


Figure 9. Average allocation rate a, generational survival rate $a\eta$, and generational survival rate with arraylets for the EEMBC benchmarks with a 32KB nursery on a 2.4 GHz Pentium.

5.1 Arraylet Pretenuring

In a generational system, arraylets have an additional enormous advantage: for most programs they allow the effective size of the nursery to be greatly increased, without increasing either its physical size or the cost of nursery collection.

The mechanism is very simple: arrayoids are allocated in the nursery, while their arraylets are allocated in the heap. As a result, the cost of evacuating the array is just the cost of moving the arrayoid into the heap: the arraylets are not moved, and the arraylet pointers in the arrayoid are unchanged, meaning that no additional pointer forwarding is incurred and the arrayoid can be block-copied.

Arraylets provide an extra level of indirection. Since we already need such a mechanism to avoid external fragmentation, using the indirection for the additional purpose of virtual nursery expansion incurs no additional cost while providing enormous benefits for applications that allocate a significant portion of their space as medium-sized or large arrays.

The "virtual evacuation" of arraylets is not free, and must be charged to the nursery collection. Thus when an arraylet is pretenured, the allocation limit for the nursery is reduced, although by much less than it would have been had the entire array had actually been allocated in the nursery.

For arrays of primitive types, the extra cost is almost zero, since in the worst case the object dies and its arraylets must be chained back into a free list.

However, for arrays of pointers, in the worst case all of the elements in the array point to objects in the nursery (whose address will change during evacuation), so each element of the array will have to be forwarded. Thus pointer arraylet allocation would have to be charged at a much higher rate. For simplicity we do not pretenure pointer arraylets.

5.2 Effectiveness of Arraylet Pre-tenuring

Figure 9 shows the effectiveness, in terms of absolute allocation rate, of arraylet pre-tenuring in our implementation. Absolute rates are shown because they determine the overall utilization that can be achieved.

Of the six benchmarks, three have low survival rates without arraylet pre-tenuring (Chess, Parallel, and Regex). Chess and Regex do not allocate arrays at all, so there can be no further benefit, but they already have very low allocation rates (both relatively and abolutely).

kXML has a moderate survival rate without arraylet pre-tenuring (about 40%), while Crypto and PNG have survival rates so high that generational collection will not be useful.

When arraylet pre-tenuring is applied, it significantly reduces the survival rate of two of the three benchmarks that allocate significant amounts of array data. The effect is particularly dramatic for Crypto, but also significant for Parallel. For PNG, the reduction is significant but the resulting surival rate may still be too high for generational collection to be profitable.

Overall, arraylet pre-tenuring has a significant effect in reducing the allocation rate and increasing the usability of generational collection. While it is not a panacea, for some programs it will provide major improvements. Most importantly, it significantly reduces the number of outliers.

6. Related Work

Generational collection for general-purpose hardware was developed by Ungar [23] for Smalltalk-80, a highly dynamic object-oriented language. Since even closures for conditionals were heap-allocated, the lifetime of most objects was extremely short and generational collection proved highly effective. Subsequently it became apparent that generational techniques were effective for a broader class of languages, especially as sufficient memory resources for larger nurseries became available.

Generational collection has proved so effective that many more sophisticated techniques have been unable to match its performance.

Moon [20] concurrently developed essentially the same technique for Lisp (called ephemeral garbage collection) using special-purpose hardware support on the Symbolics 3600. This system was not only generational but concurrent, also having hardware support for Bakers algorithm. It thus anticipates our work in some respects. However, like Baker's algorithm, it was incremental but not truly real-time.

A number of other systems have combined generational and concurrent collection. Doligez et al. [13] developed a collector for ML which exploited the large proportion of immutable objects by allocating them in independently collected nurseries. Nursery collection was synchronous and thread-local. Domani et al. [14] subsequently expanded on this basic design for a concurrent, non-compacting collector in which nursery collection was also concurrent. However, both collectors do not perform generational collection during tenured space collection, which is a fundamental requirement in our system for maintaining real-time behavior. Their system also used card marking, which is not suitable for a real-time collector due to the need to scan all cards and then scan all objects corresponding to dirty cards.

In the mid-1970's the first algorithms were developed for parallel and concurrent collection, which was viewed as a paradigmatic example of the difficulties of expressing concurrent algorithms and proving them correct [12, 22, 19]. These collectors were all *incremental update* collectors: changes to the object graph during collection were detected and the collection re-traced the graph from the modified nodes.

Yuasa [24] introduced the snapshot-style collector. Unlike incremental update collectors, Yuasa's collector operated on a virtual snapshot of the object graph at the time collection started. Yuasa's algorithm results in more floating garbage and requires a more expensive write barrier, but is better suited to real-time collection since operations by the mutator can not "undo" the work done by the collector.

Baker [7] was the first to attack the problem of real-time garbage collection. As we discussed in Section 2, his technique fundamentally suffers from using work-based, event-triggered scheduling, and from evaluating real-time properties from the point of view of the collector rather than the application. The result is fundamentally soft real-time (best effort) rather than hard real-time (guaranteed) response.

There have been many incremental and soft real-time collectors since then, exploring various aspects of the design space, such as the use of virtual memory support [2] and coarse-grained replication with a synchronous nursery [21]. However, there is no guarantee on the maximum pause time.

Cheng and Blelloch [11] described a time-triggered real-time multiprocessor replicating collector with excellent utilization, for which they introduced the minimum mutator utilization (MMU) metric, an application-oriented measure of the behavior of a concurrent collector. However, the MMU was measured rather than guaranteed, and space overheads were large.

Bacon et al. [3] built a soft real-time reference counting collector for weakly ordered multiprocessors. This collector also achieved very good MMU, but was subject to unpredictable behavior in the presence of cycles. Space overhead was low, but not guaranteed since incremental compaction was not performed.

The Metronome collector of Bacon et al. [6] was the first guaranteed hard real-time collector. This collector provided guaranteed MMU based on the the characterization of the application in terms of maximum live memory and allocation rate. Space overhead was usually comparable to that required by synchronous ("stop-the-world") collectors, due to incremental defragmentation and quantitative bounding of all sources of memory loss [5].

While most previous work on real-time collection has focused on work-based scheduling, there are some notable exceptions. In particular, Henriksson [17] implemented a Brooks-style collector [10] in which application processes are divided into two priority levels: for high-priority tasks (which are assumed to be periodic with bounded compute time and allocation requirements), memory is pre-allocated and the system is tailored to allow mutator operations to proceed quickly.

7. Conclusions

The utilization level achievable by real-time garbage collection is most fundamentally limited by the allocation rate of the program. Generational collection often reduces the effective allocation rate into the mature heap by a large margin.

We have shown how generational techniques can be applied to a real-time collector by extending the Metronome with synchronous nursery collection and *syncopation*, a technique which allows the collector to avoid exceeding its real-time bounds during temporary bursts in allocation or survival rates. When combined with multiple beat-per-measure scheduling, syncopation allows for high utilization based on average rather than peak allocation rates.

In order to determine when generational collection is advantageous, we have introduced a cost model that allows the behavior of the generational and non-generational collectors to be modeled analytically.

We have provided measurements which show the need for syncopation in practice, but also show that allocation rates stabilize very quickly (between 1 and 3 milliseconds) and that the amount of syncopated data is small.

Finally, we introduced *arraylet pre-tenuring* as a technique for increasing the effective nursery size without increasing the collection cost, and provided measurements that show that it is very effec-

tive for reducing the effective survival rate of programs that allocate significant array data.

As highly complex real-time systems become more prevalent, the capability to perform high-performance real-time garbage collection will become increasingly critical. Extending generational collection to the real-time domain will form an important part of this effort, and we are continuing to work intensively on both the theory and the implementation of such systems.

References

[1] ALPERN, B., ET AL. The Jalapeño virtual machine. *IBM Syst. J. 39*, 1 (Feb. 2000), 211–238.

[2] APPEL, A. W., ELLIS, J. R., AND LI, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices*, *23*, 7 (July), 11–20.

[3] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 92–103.

[4] BACON, D. F., CHENG, P., AND GROVE, D. Garbage collection for embedded systems. In *Proceedings of the Fourth ACM International Conference on Embedded Software* (Pisa, Italy, Sept. 2004), pp. 125–136.

[5] 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). *SIGPLAN Notices*, *38*, 7, 81–92.

[6] 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). *SIGPLAN Notices*, *38*, 1, 285–298.

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

[8] BOBROW, D. G., AND MURPHY, D. L. Structure of a LISP system using two-level storage. *Commun. ACM 10*, 3 (1967), 155–159.

[9] 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.

[10] 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.

[11] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 125–136.

[12] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-fly garbage collection: An exercise in cooperation. In *Hierarchies and Interfaces*, F. L. Bauer et al., Eds., vol. 46 of *Lecture Notes in Computer Science*. 1976, pp. 43–56.

[13] DOLIGEZ, D., AND LEROY, X. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conf. Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Jan. 1993), pp. 113–123.

[14] DOMANI, T., KOLODNER, E. K., AND PETRANK, E. A generational on-the-fly garbage collector for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, *35*, 6, 274–284.

[15] EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM. Java GrinderBench version 1.0. URL www.eembc.org, 2004.

[16] GRCEVSKI, N., KILSTRA, A., STOODLEY, K., STOODLEY, M., AND SUNDARESAN, V. Java just-in-time compiler and virtual machine improvements for server and middleware applications.

[17] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[18] JONES, R., AND LINS, R. *Garbage Collection*. John Wiley and Sons, 1996.

[19] LAMPORT, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of the 1976 International Conference on Parallel Processing* (1976), pp. 50–54.

[20] MOON, D. A. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, Aug. 1984), pp. 235–246.

[21] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.

[22] STEELE, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.

[23] UNGAR, D. M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, 1984), P. Henderson, Ed. *SIGPLAN Notices*, *19*, 5, 157–167.

[24] YUASA, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.