Generational Real-Time Garbage Collection

A Three-Part Invention for Young Objects

Daniel Frampton[1], David F. Bacon[2], Perry Cheng[2], and David Grove[2]

[1] Australian National University,
Canberra ACT, Australia
Daniel.Frampton@anu.edu.au
[2] IBM T.J. Watson Research
19 Skyline Drive, Hawthorne, NY 10952, USA
bacon,perryche,groved@us.ibm.com

Abstract. While real-time garbage collection is now available in production virtual machines, the lack of generational capability means applications with high allocation rates are subject to reduced throughput and high space overheads.

Since frequent allocation is often correlated with a high-level, object-oriented style of programming, this can force builders of real-time systems to compromise on software engineering.

We have developed a fully incremental, real-time generational collector based on a *tri-partite nursery*, which partitions the nursery into regions that are being allocated, collected, and promoted. Nursery collections are incremental, and can occur within any phase of a mature collection.

We present the design, mathematical model, and implementation of our collector in IBM's production Real-time Java virtual machine, and show both analytically and experimentally that the collector achieves real-time bounds comparable to a non-generational Metronome-style collector, while cutting memory consumption and total execution times by as much as 44% and 24% respectively.

1 Introduction

With the advent of hard real-time garbage collection [1] and its incorporation into a production virtual machine [2], Java is finally making significant inroads into domains with hard real-time concerns such as audio processing, military command-and-control, telecommunications, and financial trading systems.

The engineering and product life-cycle advantages consequent from the simplicity of programming with garbage collection, coupled with reliable real-time performance, obviate the need for low-level, error-prone techniques such as object pooling and manual memory management with scoped regions [3]. Furthermore, programmers no longer need to code time-critical portions of the system in lower-level, less secure languages like C.

E. Ernst (Ed.): ECOOP 2007, LNAI 4609, pp. 101–125, 2007.

However, previous incremental and real-time collectors have generally not included generational collection. Generational collection takes advantage of the fact that most objects die quickly, and tends to increase throughput and decrease memory requirements.

Since frequent allocation is often correlated with a high-level, object-oriented style of programming, the lack of generational collection can force builders of real-time systems to compromise the software engineering of their systems by manually recoding to allocate fewer objects.

A few generational collectors with various levels of soft- or hard-real-time behavior have been built [4,5], but they collect the nursery synchronously. This either leads to long pauses (in the order of 50 ms) or a very limited nursery size. For example, if the target maximum pause time is 1 ms and the evacuation rate is 100 MB/s, then the nursery can be no larger than 100 KB. At such small sizes the survival rate is often too high to derive much benefit from generational collection.

In this paper, we present a fully generational version of the Metronome real-time garbage collector [1] in which both the nursery and mature collections are performed incrementally, and in which the scheduling of the two types of collections is only loosely coupled. This allows nursery collection to occur at any time, including in the middle of a full-heap collection.

This generational algorithm is more complex but yields one significant advantage: the ability to size the nursery independent of the real-time bounds of the application. This allows the collector to achieve very short pause times (nominally 500 μs) and reliable real-time behavior, while using a nursery large enough to achieve low survival rates.

The fundamental innovation in our work is the use of a *tri-partite nursery:* the nursery is split into three regions. There is an *allocation* nursery into which new objects are allocated. Meanwhile the previous nursery, then known as the *collect* nursery can be collected, with live objects copied out into the mature area. For algorithmic reasons, references may exist from the heap and collect nursery to objects in the previous nursery that were previously promoted. In order to forward these references we retain this previous nursery as the *promotion* nursery.

The contributions of this work are:

- An algorithm for a fully generational real-time garbage collection in which the nursery and major collections are both incremental and can be arbitrarily interleaved.
- A tri-partite nursery which allows a nursery evacuation while the application continues to allocate into a new nursery.
- An analysis of the space bounds and mutator utilization of a generational collector in which the nursery size is elastic. Furthermore, we derive the nursery size which optimizes utilization and memory consumption.
- Measurements of applications showing that our generational collector is able to achieve comparable real-time behavior to a non-generational Metronome system, while using significantly less memory and increasing throughput.

2 Metronome Overview

This section describes the Metronome algorithm and implementation, both for background on real-time collection and for the purpose of understanding the system against which we benchmark the generational collector in Section 5.

The original Metronome system [1] was implemented in Jikes RVM (12.4ms worst-case pause, 44% MMU at 22.2ms). The model and algorithm of a generational variant with a synchronous nursery – the syncopated Metronome – has been published [4]. There was no actual implementation of the generational collector – only arraylet pre-tenuring – and a measurement of the effective survival rates with and without pre-tenuring. The conclusion was that the synchronous nursery technique could work for small embedded benchmarks with low survival rates, but would not be suitable as a general-purpose solution.

A second-generation version of the original Metronome algorithm was implemented in IBM's J9 JVM (1ms worst-case pauses, 70% MMU at 10ms), and was released as a product by IBM in 2006 [2]. This product also includes a full implementation of RTSJ, The Real-Time Specification for Java [3]. The work described in this paper is a generational system built on the IBM J9 product, without the RTSJ features.

The Metronome algorithm is described in greater detail in [1], but there are some differences in the J9 implementation. We will describe the J9 implementation but point out the aspects which differ from the original Jikes RVM implementation. Among other things, the J9 version implements the complete Java semantics including finalizers and weak/soft/phantom references, which were not supported in the original version.

The Metronome is a hard real-time incremental collector. It uses a hybrid of non-copying mark-sweep collection (in the common case) and selective copying collection (when fragmentation occurs).

The virtual machine scheduler alternates between execution of application ("mutator") threads and garbage collector threads, using predictable quanta and predictable spacing between those quanta. The system runs on uni- or multiprocessors (the original system only ran on uniprocessors), but alternation between application and collector is synchronized across processors with a barrier synchronization.

2.1 Time Based Scheduling

A key contribution of the Metronome system is that it abandons a fine grained work-based approach – such as that of Baker [6] – in favor of a *time-based* approach. The fundamental observation here is that the race between collector and mutator occurs at a relatively coarse time granularity; namely that of a collection cycle. Bursty allocation behavior in small time windows can then be amortized over the relatively long period of a complete collection cycle. A time-based scheduler interleaves mutator and collector work in small quanta at a ratio determined by the model. This ensures that the collector keeps up, but does so in a predictable manner amenable to providing the required real-time guarantees on utilization levels.

Minimum Mutator Utilization (MMU). In order to achieve correct time-based behavior, Metronome uses the *minimum mutator utilization* or MMU metric introduced by Cheng and Blelloch [7]. MMU is a measure of the worst-case utilization by the application (mutator) over a particular time window. MMU is independent of the length of particular collector pauses, since multiple short pauses grouped closely together can be as disruptive as a single long pause.

The system by default runs at an MMU of 70% over a 10 ms time window (that is, the application always receives at least 7 ms out of every 10 ms of real time). Shorter time windows and/or higher MMUs are possible depending on the characteristics of the application.

The system uses *over-sampling* and instead of interrupting the application for a single 3 ms quantum every 10 ms, it instead uses quanta whose nominal length is 500 μs, with a worst-case quantum of less than 1 ms. Over-sampling both reduces variance and increases the robustness of the schedule.

The original Metronome collector did not use over-sampling and was able to run at an MMU of 60% in a 20 ms window, with a worst-case pause of 8 ms.

2.2 Collector Design

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. As we will show subsequently, the introduction of generational collection greatly reduces the amount of floating garbage.

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 [8] implemented with a weak tricolor invariant [9]. 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.

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.

Fuzzy Snapshot. In order to maintain real-time bounds in the presence of a large number of threads, the requirement for an atomic snapshot of all roots is avoided by having the write barrier record both the old and the new pointers during the root scanning phase, instead of just the old pointer as is done by a conventional snapshot-at-the-beginning 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. A collection consists of many collector quanta.

The system uses a *lazy read barrier*. The laziness comes from the fact that references in the stack are not updated atomically when an object is moved. To ensure termination, object references written back into the heap are forwarded to current versions as they are written. As the marking phase traverses the heap, references are also forwarded to new versions. Old versions of moved objects can not be removed until after the next collection has been completed as there may still be references to them somewhere.

The original Metronome system used an *eager* read barrier which is slightly faster but requires a fixup pass over stack frames at the end of each collector quantum during the defragmentation phase. Especially on a system with many threads, this may lead to unacceptably long collector quanta.

Metronome achieves guaranteed real-time behavior 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.

3 Real-Time Generational Collection

The potential for reducing memory consumption and/or improving throughput by employing a generational collection technique [12,13] is well understood. The generational hypothesis states that most objects have very short lifetimes. A generational collector takes advantage of this by first allocating objects into a

nursery, and then employing collection techniques optimized for low survival rates to promote survivors into the next generation. In this paper we are concerned with a model of generational collection with two generations; a nursery and a mature space.

A key property for real-time generational collection is that the work required to perform a nursery collection be $O(nursery)$, not $O(heap)$. One consequence of this is that the collector must be able to discover all pointers into the nursery from the mature area without scanning the entire mature area. This is typically accomplished by using a *write barrier*, which adds overhead to each pointer write, but keeps track of all pointers from mature objects to nursery objects in a remembered set. In combination with other roots in the system, this remembered set can be used to collect the nursery without having to consider the mature space.

Previous systems have performed generational collection synchronously [4,5], but doing so links the responsiveness of the system to the worst case nursery collection time. In many applications, there are at least some time periods where the generational hypothesis does not hold. This either forces nursery sizes to be very small (low numbers of kilobytes) or worst case pause times to be quite large (tens of milliseconds).

To make generational real-time collection more widely applicable, we must (a) make nursery collection incremental, and (b) allow nursery collections to occur at any point during a mature collection cycle. Achieving both of these design goals decouples worst case pause time from nursery size, enabling the nursery to be sized to obtain the low survival rates critical for effective generational collection.

The rest of this section outlines the key challenges in incremental nursery collection and how our system addresses them. This is not a complete description of our generational algorithm; but it does cover all of the key extensions necessary to build an incremental generational collector on top of the base Metronome system. The next subsection describes how the tri-partite nursery enables the mutator to continue allocating while a nursery collection is in progress. The second subsection describes the techniques used to collect the nursery, including the write barriers that are used to preserve the nursery root set. The final subsection discusses interactions that arise when a nursery collection occurs concurrently with a mature space collection.

3.1 Tri-partite Nursery

The fundamental goal of our algorithm is to allow the mutators to continue executing – *and therefore allocating* – while we are collecting the nursery. In order to satisfy this requirement, while retaining a reasonable model of the system, we begin allocating new objects into a separate nursery area while we perform the collection of the previous nursery. We call this new nursery the *alloc* nursery, and the nursery being collected the *collect* nursery.

Unlike the previous synchronous generational Metronome [4], the alloc nursery does not have a fixed size. Instead, it continues to grow via mutator allocation

actions until it is both desirable and possible to begin the next nursery collection cycle. It is desirable to initiate a nursery collection once a certain amount of allocation – the *nursery trigger* – has occurred. However, if the previous nursery collection is still in progress when the nursery trigger is hit, the new nursery collection must be deferred until the prior nursery collection completes. During this time interval, the mutator can continue to allocate into the alloc nursery. The nursery trigger is a system parameter and can be varied to trade-off survival rate with memory consumption (see Section 4). The elasticity of the nursery size allows the system to smoothly absorb short-term spikes in the allocation rate, without resorting to *flood-gating*: direct allocation into the mature area.

Since the nursery collection is incremental, the mutator is free to create new pointers within the system. As demonstrated in detail in the following sections, this leads to the requirement to retain each nursery until the nursery after it has been collected. We call a nursery at this point in the lifecycle the *promote* nursery. A promote nursery contains no active object *data*, but simply forwarding pointers, or indirections to objects that have been promoted to the mature space.

All nursery pages are allocated out of the single global pool of pages shared with the mature space. This facilitates both the logical pre-tenuring of arraylets [4] and the development of a simple model of the system. Allocation into nursery pages is performed using a simple bump pointer. As the surviving objects are promoted into the mature space, they will be relocated to an appropriate size-segregated page.

3.2 Incremental Generational Collection

As with other generational approaches, we use a write barrier that checks on the fast path if a pointer is being created from a mature object to the nursery. Figure 1 shows the fast path of the barrier, including a call to the slow path when the collector is tracing. This part of the barrier supports incremental tracing, and, from the perspective of the mutator, comes at no additional cost over the base system as the same technique is used for both nursery and full heap traces.

```
write_barrier (source: OBJECT, slot: ADDRESS, target: OBJECT) {
    target = forward(target) // Ensure forwarded

    if (isMature(source))
        if (isNursery(target))
            call slow_path

    if (collector_tracing)
        call slow_path
}
```

Fig. 1. Write barrier pseudo-code

Throughout the detailed description of the nursery collection we use the following notation:

M: The mature region.
N_k: The kth nursery region.
REM_k: This is the remset that is processed when collecting N_k. It is filled during the period of time that N_k is the active *alloc* nursery.
$ROOT_k$: The set of roots that was captured at the start of collecting N_k. While we have an extension that allows the roots to be captured incrementally, for simplicity we describe the algorithm as if there was an atomic root snapshot.

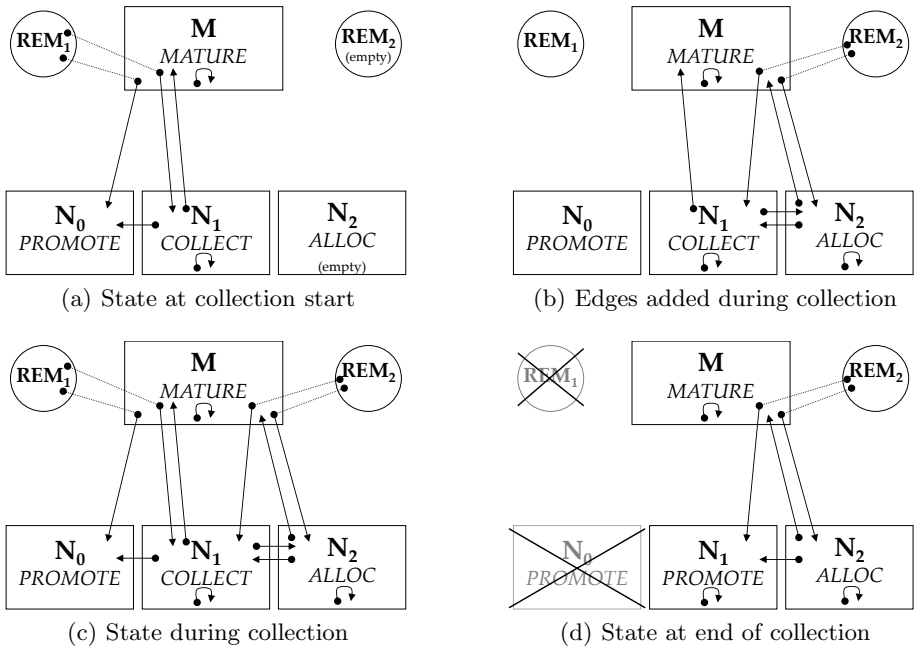


Fig. 2. Heap reference state and changes during a nursery collection

Consider the state of the system at the start of a collection, N_1 as shown in Figure 2a. For the initial nursery collection the *promote* nursery will be empty. We also have a remset REM_1 which will capture all references created from $M \rightarrow N_1$, and as demonstrated later, also any pointers into N_0 that were created during the collection of N_0.

As we begin to collect N_1 we perform the following steps atomically:

1. Take the remembered set REM_1 containing all references from $M \rightarrow N_1$ (and possibly $M \rightarrow N_0$),
2. Take the root set $ROOT_1$ (from stacks and other VM structures),

3. Switch all mutators to begin allocating into N_2
4. Switch all mutators to begin contributing remset entries into REM_2.
5. Turn on the tracing barrier to ensure the nursery is traced consistently.

Mutators are then allowed to continue running while N_1 is collected incrementally. It is clear that the union of REM_1 and $ROOT_1$ provide us with a complete *snapshot* of N_1. Therefore, *all* live objects in N_1 are transitively reachable from this set. Through the protection provided by the tracing barrier, this allows us to safely and completely collect the nursery. Note that all references created from N_2 into N_1 must have been obtained from somewhere captured by the snapshot. This means that it is not necessary to trace through N_2 during the collection of N_1. Figure 2b shows the references that can be *created* in mutator intervals that occur during a nursery collection. All pointers created to the mature area, or contained within an individual area will never be required to perform a nursery collection. They are shown on the figures for completeness.

The interesting references that can be created during collection are thus:

$\mathbf{M \rightarrow N_1}$: We include these in REM_2. We already have a complete snapshot for N_1 in $(REM_1, ROOT_1)$. As we need to retain N_1 to deal with the unbarriered pointers from $N_2 \rightarrow N_1$ we may simply leave these values here to be updated at the next collection.

$\mathbf{M \rightarrow N_2}$: We include these in REM_2: these are essentially the *normal* generational remembered set entries.

$\mathbf{N_1 \rightarrow N_2}$: These references are not write barriered. As the objects are promoted into the mature space, we will add appropriate entries to REM_2. The entries created on promotion correspond to the case above of $M \rightarrow N_2$.

$\mathbf{N_2 \rightarrow N_1}$: For objects that remain live, these references can only be discovered at the next collection (N_2). This is what requires us to retain a *promote* nursery. These references are not required to find live objects in N_1 as we have a complete snapshot for N_1 in $(REM_1, ROOT_1)$.

Figure 2c shows all references that may exist within the heap during collection. Once collection has been completed, the remset REM_1 will have been completely drained, and all objects (transitively) reachable from the mature space will have been promoted. In addition, as all references that may exist to N_0 would have existed solely in REM_1, there are now no references into N_0 and the space can be reclaimed. N_1 then becomes the *promote* nursery. This state is shown in Figure 2d. From this point the only information remaining in the promote nursery is the forwarding pointer information for promoted objects. The space is essentially closed, as all live objects have been identified and promoted.

Figure 3 shows the pointers that can be created during mutator intervals *outside* of a collection. These are simply pointers between M and N_2 (the alloc nursery), with all pointers from $M \rightarrow N_2$ captured in REM_2.

Note that after incrementing each of the indices, the state is as shown in Figure 2a when the next collection commences.

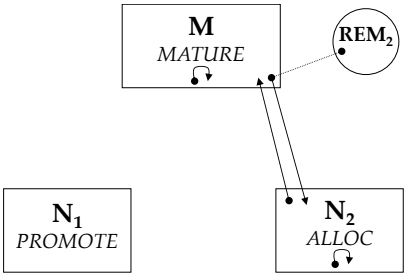


Fig. 3. Edges added outside of a nursery collection

3.3 Mature/Nursery Interaction

While the individual techniques for collecting both nursery and mature space have been shown to be correct in isolation, additional complexity is involved in combining them. In order to provide the necessary real-time guarantees, the ability for a nursery collection to occur must not be interfered with by the mature collection process. To achieve this, the mature collection leaves the system in a state where a nursery collection is possible at the end of each and every mature increment. There is no inverse requirement as the nursery collections will leave sufficient time to perform mature collection, unless the application and/or collector behavior has been incorrectly parameterized. The additional burden placed on the mature and nursery collectors must also be carefully controlled to ensure that no mature related work is performed by the nursery collector, and that the amount of additional work being performed is acceptable.

Objects are allocated with a two-field color. The color is obtained from the allocating thread and changes as the thread is scanned at the beginning of each nursery and mature collection. One field indicates the nursery epoch, while the other indicates the mature epoch. The importance of these will be understood as the following problems are discussed.

Nursery collecting out from under mature space. It is possible for objects that are live in a mature collection to be garbage from the perspective of a subsequent nursery collection. For this reason, during a mature collection, a nursery collection must keep alive the portion of the nursery that is part of the executing mature collection's snapshot. Any previously written references from the mature space to the nursery will already be captured in a remembered set. References that are subsequently lost to the nursery will be captured by the nursery's Yuasa barrier. The references to the nursery that were lost before the nursery collection began are those of interest, and these are the nursery references on the mature collection's Yuasa barrier.

The nursery treats these values as additional roots during its collection. The mature space is required to maintain nursery references separately. This avoids the nursery collector performing any mature space bounded work.

Promoting objects in the appropriate state. It is important that the nursery promotes objects into the mature space in a consistent manner. If, for example, the nursery promotes objects into the mature space as unmarked after tracing is complete, the mature space may sweep up live objects. Similarly, if the nursery promotes objects as live during tracing, references from these objects into the mature space may be missed and cause dangling pointers within the mature space.

This is the motivation for all objects being allocated with a mature epoch bit set. All objects that were allocated before the collection began will not have this bit set, and all objects allocated after the collection began will. When the nursery visits an object, it can use this bit to determine the appropriate mark state to promote with. If the object is in the previous epoch, it will either be marked by the mature collector or, interestingly, can be left as harmless garbage. If the object is in the current epoch, it is promoted as marked – which is equivalent to the allocate-black property of many snapshot collectors.

Sweeping out from under the nursery. The nursery collector maintains a remembered set of references into the nursery from the mature space. If the mature objects containing those references die, then the nursery collector would be processing garbage data looking for roots for its collection. To avoid this, the mature collector is required to sweep the remembered set, removing any references from dead objects. As these objects are garbage, any references from them into the nursery need not be traced. Additionally, when the mature space is defragmented, the nursery remset entries must be forwarded to maintain freshness.

4 Analytical Model

Intuitively, a generational collector is more efficient than a full-heap collector because processing an area in which there are many dead objects allows reclamation of more space for a given amount of GC work. However, when the survival rate η is high or even comparable to the survival rate of a full-heap collection, the generational variant will fare worse because of the cost of determining what is live as well as the actual copying. In this section, we model the behavior of the generational collector and compare it to that of the original Metronome collector and the syncopated Metronome collector.

4.1 Definitions

We begin by characterizing the garbage collector itself by the following parameters:

- 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);

As mentioned earlier, generational collectors may be a net loss because inherently, $R_T > R_N$ (because tracing is faster than tracing together with copying). The application is characterized by the following parameters:

- a is the allocation rate (bytes/second) in mutator time (that is, allocation rate ignoring the times when garbage collection is active);
- m is the maximum live memory of the mutator (bytes);
- $\eta(N)$ is the survival rate in the nursery. Specifically, it is the portion of the objects (by bytes) that is live (taking into account the generational barrier) of the last N allocated bytes. This function is monotonically decreasing in N.

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 [7] in each Δt;

4.2 Steady-State Assumption and Time Conversion

The allocation rate a and the survival ratio η in fact can vary considerably as the application runs. For the time being we will consider the case when they are smooth. However, since the nursery size varies dynamically as a central aspect of this algorithm, we model it dynamically. As in previous Metronome collectors, modeling relies on being able to convert from mutator time to GC time. 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.

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

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

$$u = \frac{\gamma}{1+\gamma} \tag{2}$$

From the above parameters, we can then derive the overall space consumption of the system. Fundamentally, for all real-time collectors, the space requirements depend on the amount of extra memory that is allocated during the time when incremental collection is being performed and the mutator is continuing to run. Thus:

- s is the space requirement (bytes) of the application in our collector, and
- e is the extra space allocated by the mutator over the course of a full-heap collection.

We will review bounds for s and e for previous collectors and then show how they relate to our collector.

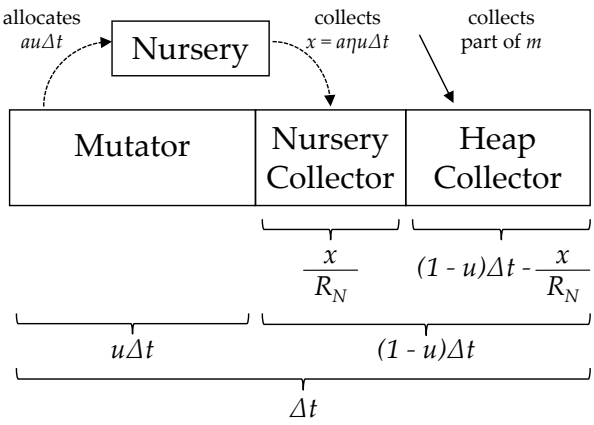


Fig. 4. Time dilation due to generational collection causes additional allocation during a major heap collection, but attenuates all allocation by the survival rate η

4.3 Bounds for Non-generational Metronome Collectors

In the absence of generational collection, the extra space e_M for Metronome (as described in [1]) is

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

which is the allocation rate multiplied by the time required to perform a collection, converted into mutator time by the γ factor.

Freeing an object in Metronome-style collectors may take as many as three collections: (1) to collect the object; (2) 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) because we may need to relocate the object in order to make use of its space. The first two aspects are common to incremental collectors; the third is specific to Metronome's approach to defragmentation.

$$s_M = (m + 3e_M) \cdot (1 + \rho) \tag{4}$$

In other words, the maximum space required is the base memory plus three times the extra memory allocated during collection multiplied by the amount of fragmentation.

4.4 Bounds for Our Generational Collector

When performing generational collection, the time spent collecting the nursery reduces the rate of progress of the full heap collection. This in turn means that the mutator performs more allocation during collection. However, with generational collection the allocation into the mature area is also attenuated by the survival rate $\eta(N)$. This effect is shown in Figure 4, expressed by the following

equations, in which we define the generational *dilation factor* δ and the corresponding extra space e_G under generational collection:

$$\delta = 1 - \frac{a\eta(N)}{R_N} \cdot \gamma \tag{5}$$

$$e_G = \frac{a\eta(N)\gamma}{\delta} \cdot \left(\frac{m}{R_T} + \frac{s}{R_S} \right) \tag{6}$$

Since our generational collector is fully incremental, we can maintain real-time behavior without limiting the size of the nursery, and therefore use a nursery size which is best suited to the survival rate of the application. However, this flexibility also leads to additional complexities in determining what that size should be.

Since $\eta(N)$ is monotonically decreasing and low η values are crucial to the success of generational collection, let us consider what happens as the nursery size varies. When the nursery size is very small, the collector will spend all its time performing nursery collections because the low survival rate leads to very unproductive nursery collections. In fact, because of the way in which the nursery grows, generational collection will not complete until the nursery grows in size until exactly all of the collection time is spent solely on minor collections at which point

$$\frac{N \cdot \eta(N)}{R_N} = \gamma \frac{N}{a} \tag{7}$$

In other words, N grows until it reaches a minimum tenable size N_{min}:

$$\eta(N_{min}) = \frac{\gamma \cdot R_N}{a} \tag{8}$$

Note that this requires that $R_N > a\eta(N)\gamma$.

When the nursery size is set above this threshold, major collections are given an opportunity to complete and there is a bound on the memory consumption. If the nursery is set arbitrarily large, overall memory consumption increases as the nursery dominates the mature space in size. Between these two extremes is a nursery size which minimizes the overall heap consumption. In order to compute this point and to compare the generational system against the non-generational version, we need to to compute the space bounds of the system.

Of course, the generational version has the additional space cost of the tripartite nursery. As a result, the space requirement of our collector paired with a given application is

$$s_G = (m + 3e_G) \cdot (1 + \rho) + 3N \tag{9}$$

The $(1 + \rho)$ factor is notably absent in the $3N$ term because of the lack of fragmentation in the nurseries. As we pointed out before, the nursery will grow in size until it passes the threshold of equation 8. If the nursery is larger than this crossover point, the heuristic will not grow the nursery in size anymore.

However, continuing to grow the nursery in size will actually diminish overall heap consumption. This can be seen because just above the crossover point, the term δ is infinitesimally positive so that s_G is arbitrarily large. Similarly, when N approaches infinity, s_G is arbitrarily large. Thus, if we hold utilization constant (because it is a target), there must exist, by continuity, a globally minimal overall heap size for some nursery size. Inverting the function to express utilization in terms of s_G gives the achievable utilization for a particular overall heap size.

Note that we are making a steady-state assumption about $\eta(N)$. Since we are collecting the nursery itself incrementally and therefore handle a wide range of nursery sizes, this is reasonable for a large class of real programs. However, there is also a class of programs that have a setup phase which precedes steady-state (or "mission") phase. For such programs the steady-state assumption, when applied to the entire program, may produce overly large nurseries. We will study an example of such a program in Section 5.4. This effect is also present in non-generational real-time collectors, but is exacerbated in generational collectors. For both types of systems, it is desirable to allow the application to explicitly delineate the setup and mission phases, and to either allow real-time bounds to be violated during the setup phase in favor of reduced memory consumption, or to perform a (potentially synchronous) memory compaction between the two phases.

4.5 Comparison with Syncopation

Generational collection in a Metronome-style collector was previously described using a technique called *Syncopation* [4]. Syncopation uses *synchronous* collection of the nursery combined with *flood-gating* — direct allocation into the mature space — when allocation and survival rates are too high for synchronous collection to be performed without violating real-time bounds.

However, with syncopation the nursery size N was not really variable, since the synchronous nursery collection places severe bounds on real-time behavior. With such small nurseries, real-world programs almost always contain spikes in the survival rate such that for all practical N, $\eta(N) \to 1$. Therefore it was generally necessary to use the largest possible nursery size such that

$$\frac{N}{R_N} = (1-u)\Delta t \tag{10}$$

$$N = (1-u)\Delta t R_N \tag{11}$$

The time dilation and extra space calculations then become simpler, such that

$$\delta' = 1 - \frac{N}{R_N} \cdot \gamma \tag{12}$$

$$e_S = \frac{a\gamma}{\delta'} \cdot \left(\frac{m}{R_T} + \frac{s}{R_S} \right) \tag{13}$$

and the space bound for synchronous nursery collection is

$$s_S = (m + 3e_S) \cdot (1+\rho) + (1-u)\Delta t R_N \tag{14}$$

Although there is no factor of 3 multiplier on the nursery as for our generational collector (equation 9), the higher survival rates incurred by the much smaller nurseries mean that the space consumption in the mature space increases significantly.

5 Experimental Evaluation

We have implemented our generational algorithm as a modification[1] to the IBM WebSphere Real Time Java virtual machine [2], which uses the non-generational Metronome-based algorithm described in Section 2. Both collectors support the complete Java semantics, including finalization and weak/soft/phantom references.

The syncopated Metronome, discussed in Section 4, is not experimentally compared. The nursery sizes required to achieve low survival rates on non-embedded applications – in the order of 1MB for SPECjvm98 – would incur pauses of at least an order of magnitude beyond the worst-case latencies for the other systems.

All experiments were run on an IBM Intellistation A Pro workstation with dual AMD Opteron 250 processors running at 2.4 GHz with a 1 MB L2 data cache. Total system memory was 4 GB RAM.

The operating system was IBM's real-time version of Linux[2] based on Red Hat Enterprise Linux 4. This includes a number of modifications to reduce latency, in particular the `PREEMPT_RT` patch with modifications for multi-core/multi-processor systems.

We begin our evaluation with a performance comparison of the generational and non-generational systems across a range of benchmarks. We then demonstrate the effectiveness of the dynamic nursery size at coping with short bursts of allocation. Selecting a highly generational benchmark, `jess`, we show the importance of large nursery sizes made possible through incremental nursery collection. We then highlight the difficulties in fairly comparing real-time collectors by observing differences between startup and steady-state behavior.

As we are interested in comparing the collector performance of two alternatives, we use a modified *second run* methodology. This methodology involves invoking a benchmark twice within a single JVM invocation, the first *warmup* run performs compilation and optimization, while results are gathered from a second *measurement* run. This methodology better isolates the performance differences due to the collector.

The JIT implementation in our system is not real-time, so it is necessary to disable it during the measurement run. Between the warmup and measurement

[1] In addition to adding generational capabilities, we also disabled support for the Real-Time Specification for Java (RTSJ) standard [3] and enabled defragmentation in both the base and generational configurations of the JVM. Therefore, the performance results for our base system are not directly comparable to the product.

[2] `ftp://linuxpatch.ncsa.uiuc.edu/rt-linux/rhel4u2/R1/rtlinux-src-2006-08-30-r541.tar.bz2`

runs we disable the JIT by calling `java.lang.Compiler.disable`, and pause to allow the compilation queue to drain. IBM's real-time JVM also includes an ahead-of-time compiler which could be used to factor out JIT interference, but the generated code is slower than that produced by the JIT and therefore – since the mutator is running slower – does not stress the garbage collector as much.

5.1 Generational vs. Non-generational Comparison

We performed a comparison of the generational and non-generational Metronome using the SPECjvm98 and DaCapo [11] benchmark suites. A summary of the results is shown in Table 1. For the baseline (non-generational) system we show the total run time, and the portions of time spent in mutator and collector (GC). We also show peak memory use, average memory use, and the achieved MMU (based on a target of 70%). For the generational collector, we show values relative to the baseline to facilitate comparison. Figures reported are relative times, the fraction of garbage collection time spent in nursery collection, the relative memory consumption figures, and the achieved MMU.

Table 1. Comparison of non-generational with generational Metronome collector

		Full Heap Metronome						**Generational Metronome**						
	Trigger	Time (s)			Mem. (MB)			Relative Time			Nur.	Rel. Mem		
Bench.	(MB)	Total	Mut.	GC	Peak	Avg.	MMU	Total	Mut.	GC	Frac.	Peak	Avg.	MMU
compress	24, 2	8.99	8.162	0.126	28.77	14.45	70%	0.99	0.98	1.87	84%	1.00	1.01	69%
jess	8, 2	8.162	6.526	1.636	12.16	8.20	69%	0.84	0.94	0.43	77%	0.69	0.80	69%
rayttrace	16, 2	4.501	3.433	1.067	29.28	19.98	69%	0.76	0.90	0.28	81%	0.99	0.55	70%
db	24, 2	13.18	12.38	0.798	32.62	20.17	67%	1.00	1.00	0.89	57%	1.09	1.04	68%
javac	24, 2	6.365	4.99	1.375	49.27	32.78	67%	1.14	1.09	1.35	92%	1.70	2.03	68%
mpegaudio	8, 2	10.24	10.24	0	2.47	2.41	100%	1.01	1.01	1.00	NA	0.78	0.77	100%
mtrt	24, 2	3.126	2.388	0.738	82.97	46.87	69%	0.88	0.97	0.61	75%	0.93	0.55	67%
jack	8, 2	4.222	3.633	0.588	10.48	6.90	69%	0.92	0.97	0.64	81%	0.82	0.90	70%
antlr	20, 4	5.426	5.063	0.362	23.64	14.26	69%	0.94	0.91	1.33	59%	1.03	1.04	68%
bloat	24, 4	30.83	26.99	3.831	45.62	20.24	69%	0.88	0.94	0.47	92%	0.56	0.83	69%
chart	36, 4	159.8	147.5	12.24	51.14	25.55	67%	0.99	1.06	0.24	80%	0.80	1.10	67%
eclipse	64, 8	90.14	77.47	12.67	80.86	66.66	56%	0.95	0.98	0.78	65%	1.23	0.75	67%
fop	24, 4	3.210	2.857	0.353	27.22	22.09	70%	1.00	1.00	0.94	82%	0.89	0.83	69%
hsqldb	144, 16	4.753	4.303	0.450	158.48	116.29	70%	1.47	1.24	3.71	100%	1.11	0.89	63%
jython	20, 4	22.44	18.53	3.911	46.72	24.89	67%	0.93	0.99	0.69	69%	0.76	0.68	63%
luindex	20, 4	17.71	16.59	1.118	21.38	14.86	68%	1.06	1.03	1.41	79%	1.02	1.04	69%
lusearch	36, 8	17.29	13.18	4.114	48.75	34.79	68%	0.97	1.00	0.88	35%	1.11	0.98	66%
pmd	48, 4	30.34	24.98	5.364	71.30	47.00	68%	0.98	0.88	1.42	89%	2.48	1.68	66%
xalan	128, 12	12.49	11.43	1.051	136.86	87.49	64%	1.16	1.11	1.80	71%	1.00	1.04	68%
geomean								0.983	0.997	0.883		0.995	0.93	

For each benchmark, the first column reports the full heap and nursery triggers used for that benchmark. The full heap triggers are based on each program's steady-state allocation rate and maximum live memory size; the nursery trigger was selected by evaluating a range of possibilities (512KB through 16MB) and picking the trigger that enabled the best time/space performance. Note that these are *triggers* and not *heap sizes*. Because of the nature of incremental collection, for a given set of parameters the system may require differing amounts

of memory to run without violating its real-time requirements. When comparing stop-the-world collectors, a simpler methodology may be used in which the heap size is fixed and the resulting throughput is measured. With a real-time collector there is an additional degree of freedom, so the comparison is more complex, with an inter-relationship between total run time, total memory usage, and MMU.

The reported memory size includes both the size of the heap and the size of the nursery. This is both to make a fair comparison and it also reflects the nature of our system, in which nursery pages and heap pages are intermingled in physical memory. Note that the full heap collection trigger is with respect to this total usage – that is, it includes the memory being consumed by the nursery.

As predicted by the analytic model presented in Section 4, generational collection is better for many, but not all benchmarks. Overall, it reduces both time and space, with most of the speedup coming from reduction in time spent in the collector. However, time varies from a 24% speedup on `raytrace` to a 47% slowdown on `hsqldb` and space varies from a 44% reduction on `bloat` to a 148% increase on `hsqldb`. Real-time performance (MMU) is essentially the same, with the largest variation being 7% on `hsqldb`. Many benchmarks have short periods where they which they exhibit non-generational behavior, leading to peak memory usage higher than the non-generational system, while average usage across the whole execution is lower. An example is `eclipse`, where the generational system has a peak usage 25% higher, but average memory use is just 75% of the base system over the entire run. Overall, for programs that are at least somewhat generational in their memory allocation and usage patterns, the generational collector offered significant performance benefits. Significant degradations correlated with non-generational memory usage patterns.

5.2 Dynamic Nursery Size

The use of a single pool of pages for both the nursery and the heap, and the ability of the nursery to temporarily consume more than its trigger size, allows our collector to gracefully handle temporary spikes in the allocation rate. Table 2 shows the minimum, mean, and maximum nursery sizes for each benchmark (`mpegaudio` performs so little allocation that it never fills a 2MB nursery, so there is no data for it). Many of the benchmarks do in fact have a maximum nursery size three or more times as large as the nursery trigger, and in the case of `mtrt` the nursery is 15 times as large as the trigger size. As the nursery trigger gets larger, this effect is less dramatic but can still be seen to some degree on most of the benchmarks. These variations show that the dynamically sized nursery is highly effective at absorbing short-term allocation bursts, while maintaining overall space bounds and real-time behavior.

5.3 Parameterization Studies

In Section 4, we discussed analytically the effect of varying the nursery size on total memory consumption. Figure 5 shows the overall performance of the `jess`

Table 2. Dynamic Variation in Nursery Sizes Absorbs Uneven Allocation Rates

Benchmark	Trigger	Mean	Maximum	Std. Dev.
_201_compress	2.0	5.3	6.0	1.37
_202_jess	2.0	2.0	2.4	0.02
_205_raytrace	2.0	2.2	9.6	0.85
_209_db	2.0	2.2	6.2	0.67
_213_javac	2.0	2.9	7.1	1.31
_222_mpegaudio	—	—	—	—
_227_mtrt	2.0	2.8	31.7	3.96
_228_jack	2.0	2.0	2.1	0.01
antlr	4.0	4.05	4.31	0.05
bloat	4.0	4.05	4.31	0.02
chart	4.0	4.05	4.39	0.04
eclipse	8.0	8.04	8.97	0.07
fop	4.0	4.05	4.17	0.04
hsqldb	16.0	25.30	39.97	8.45
jython	4.0	4.11	5.48	0.20
luindex	4.0	4.03	4.06	0.004
lusearch	8	8.07	8.36	0.07
pmd	4.0	5.28	17.39	2.61
xalan	12.0	12.04	12.05	0.005

benchmark as we vary the nursery size from 256KB to 3072KB. We choose `jess` as it is highly generational and therefore allows us to clearly see the effect of altering the nursery trigger. Non-generational programs are likely to perform poorly on all feasible nursery sizes. Both the time and space measurements are point-wise normalized against the non-generational system. The most dramatic effect is that at low nursery sizes, the memory usage spikes upwards (beyond the range of the graph) as predicted by divergence condition in equation 7. Somewhere around a 512KB nursery size, the memory consumption of the generational and non-generational system are similar. Around 1.5MB, further increases in the nursery size do not improve the efficiency of the nursery collections so that the mature space does not decrease fast enough to compensate for the triple increase in space that the $3N$ term charges so that memory consumption begins to increase. Note that total time spent in nursery collections also monotonically decreases as we increase the nursery size as the total amount of data that is promoted decreases as $\eta(N)$ decreases. Mutator time is fairly consistent across nursery sizes and the shape of the total execution time mutedly follows the shape of the GC Time.

Figure 6 shows the dynamic behavior of memory consumption and mutator utilization of the `jess` benchmark when the nursery size is set to 3 different regimes. Generally, as we increase the nursery size, the overall efficiency of collection improves and total time spent in garbage collection decreases. For the very low nursery size of 256KB in sub-figure (a), all the time is spent in minor collections and the nursery is barely big enough for even a minor collection to complete. Consequently overall memory consumption is unbounded as the mature space keeps growing. The thick band shows that the utilization is always oscillating between 72% to 85% indicating that the GC has no breathing room at all. When the nursery trigger size is doubled as in sub-figure (b), the nursery

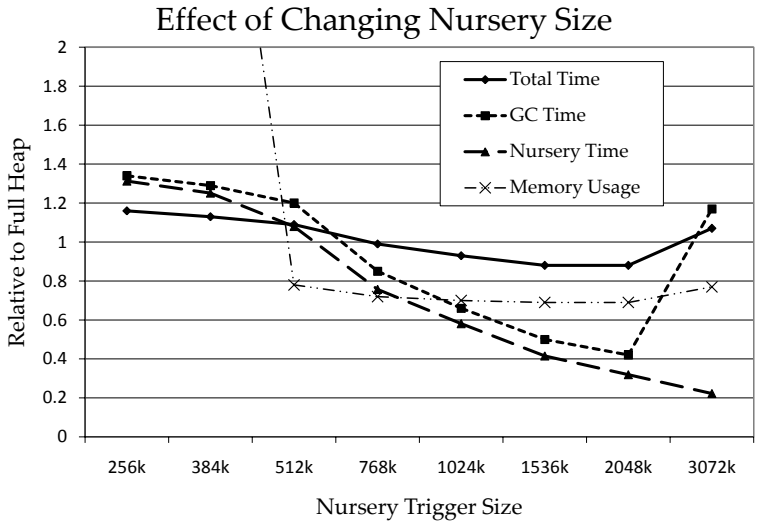
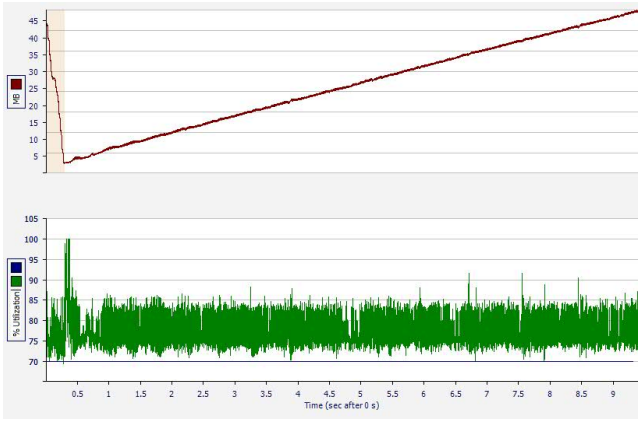


Fig. 5. Effect of changing nursery size for _202_jess with an 8m mature trigger

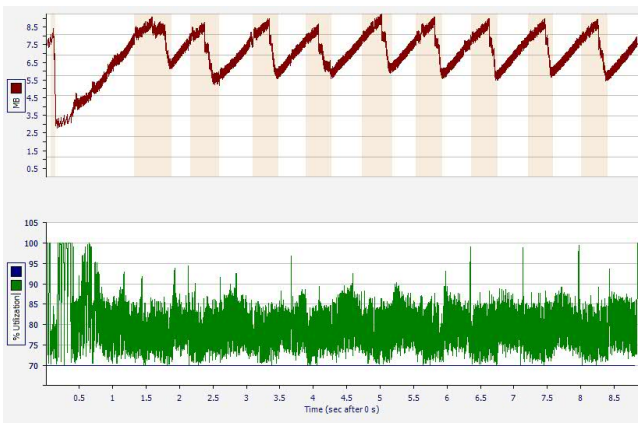
collections complete before the subsequent nursery is filled, allowing major collection work to occur and leading to a bounded mature heap size of around 9MB. Utilization is not as constant as the GC does not have to work as hard so that mutator utilization is occasionally around 90% and each major GC generally take half a second. When nursery size is further increased as in sub-figure (c), minor collections complete early enough that a large fraction of overall collection time can be spent in major collection that each major collection takes only a tenth of a second. Often, there are no active collections(either major or minor) so that overall utilization reaches 100% and averages around 85%. Because overall efficiency is improved, the heap consumption is 8.25MB and is actually lower even though the nursery is larger.

5.4 Startup vs. Steady State Behavior

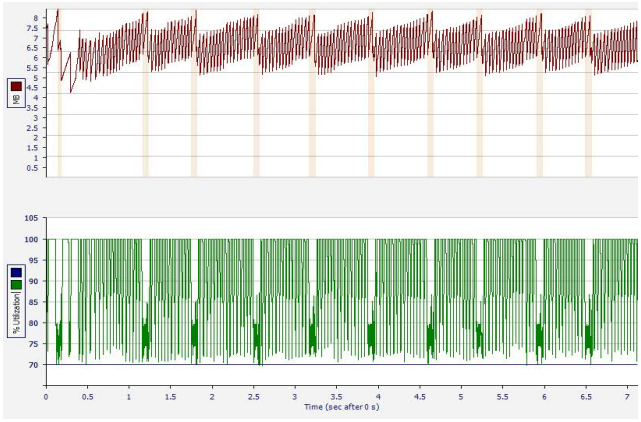
Figure 7 shows the memory consumption of `pseudojbb` under both systems. This benchmark begins by setting up several large data structures and then runs many transactions each of which slightly modify the pre-existing large data structures. In the first phase, both the allocation rate and the survival rate is high. As a result, the generational system's nursery is unable to absorb the allocation completely and must grow. During this period, as objects are promoted, there are often two copies of portions of the long-lived data structures. In this phase, the nurseries cause the memory consumption to be 45% higher than that of the non-generational system. However, once we reach the "mission" phase of



(a) 256KB Nursery Trigger

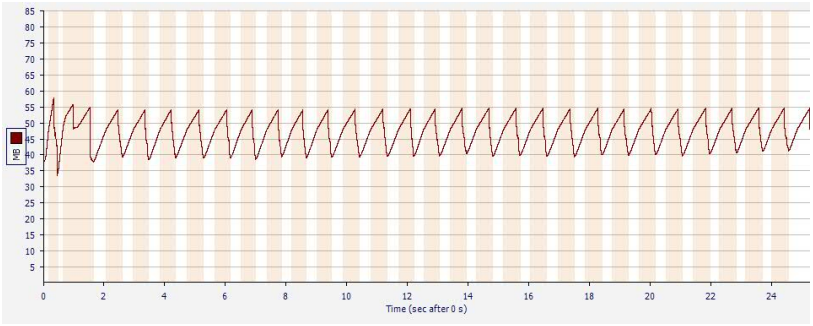


(b) 512KB Nursery Trigger

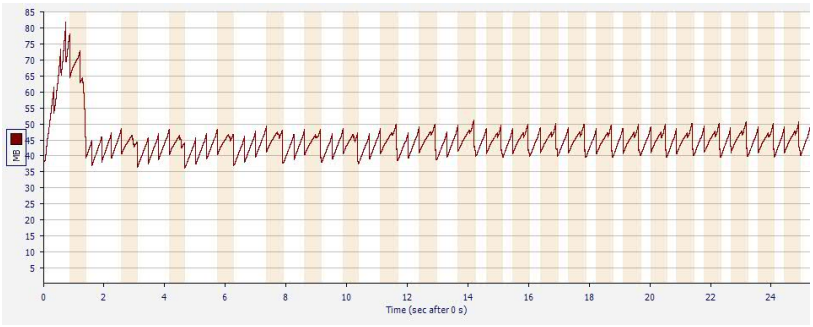


(c) 2MB Nursery Trigger

Fig. 6. Performance of _202_jess with varying nursery trigger and 8m mature trigger



(a) Non-generational



(b) Generational with 8 MB Nursery Trigger

Fig. 7. Memory usage over time of pseudojbb under non-generational and generational collection

the application (about 1.8 seconds into the run), the greater efficiency of the generational system dominates, resulting in a 10% reduction in space consumption and less time spent in garbage collection.

6 Related Work

Generational collectors were concurrently introduced by Ungar [12] and Moon [13] and have proven to be so effective that many more sophisticated partial-heap techniques have been unable to match its performance.

A number of other systems have combined generational and concurrent collection. Doligez et al. [5] 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.

Yuasa [8] introduced a snapshot-style incremental 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 [6] 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 [15] and coarse-grained replication with a synchronous nursery [16]. However, there is no guarantee on the maximum pause time.

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 (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.

Cheng and Blelloch [7] described a time-triggered real-time multiprocessor replicating collector with excellent utilization, for which they introduced the minimum mutator utilization (MMU) metric, a generic application-oriented measure of the behavior of a concurrent collector. However, MMU was measured rather than guaranteed, and space overheads were large. A generational variant was presented but the replication-based techniques and need for an atomic flip meant pointer arrays doubled in size as each logical slot required two physical slots.

The Metronome collector of Bacon et al. [1] 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 [18].

Bacon et al. [4] introduced a real-time generational collector that used a synchronous ("stop-the-world") nursery collector [4]. Though this works well for embedded benchmarks with nurseries in the tens of kilobytes, larger nurseries quickly push maximum pause times into the tens of milliseconds. This effect is exacerbated by the need to "over-sample" (collecting the nursery multiple times within a single MMU quantum) in order to avoid pathological behavior during allocation rate spikes. This forces the use of smaller nurseries, which increases survival rate and lowers the effectiveness of generational collection.

7 Conclusion

We have presented a new algorithm for performing generational collection incrementally in real-time, based on a tri-partite nursery which overlaps allocation, collection, and defragmentation. Generational collection can be interleaved with incremental real-time collection of the mature space at any point. The resulting algorithm allows the use of large nurseries that lead to low survival rates, and yet is capable of achieving sub-millisecond latencies and high worst-case utilization.

We have implemented this new algorithm in a product-based real-time Java virtual machine, and evaluated analytically and experimentally the situations under which our generational collector is superior to a non-generational real-time collector. Programs with inherently non-generational behavior and programs whose setup phase includes unusually high survival and allocation rates, will require more space to achieve the corresponding real-time bounds. However, the results show that for most programs, generational collection achieves comparable real-time bounds while leading to an improvement in space consumption, throughput, or both.

References

1. Bacon, D.F., Cheng, P., 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, pp. 285–298. ACM Press, New York (2003)
2. IBM Corporation: WebSphere Real Time Java virtual machine (August 2006) `http://www.ibm.com/software/webservers/realtime`
3. Bollella, G., et al,: The Real-Time Specification for Java. In: The Java Series, Addison-Wesley, UK (2000)
4. Bacon, D.F., Cheng, P., Grove, D., Vechev, M.T.: Syncopation: generational real-time garbage collection in the Metronome. In: Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 183–192. Chicago, Illinois (June 2005)
5. Doligez, D., 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, pp. 113–123. ACM Press, New York (1993)
6. Baker, H.G.: List processing in real-time on a serial computer. Communications of the ACM 21(4), 280–294 (1978)
7. Cheng, P., Blelloch, G.: A parallel, real-time garbage collector. In: Proc. of the SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, Utah, pp. 125–136 (June 2001)
8. Yuasa, T.: Real-time garbage collection on general-purpose machines. Journal of Systems and Software 11(3), 181–198 (1990)
9. Jones, R., Lins, R.: Garbage Collection. John Wiley and Sons, Chichester (1996)
10. Brooks, R.A.: Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In: Steele, G.L. (ed.) Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp. 256–262. ACM Press, New York (1984)

11. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Eliot, J., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 169–190. ACM Press, New York (2006)
12. Ungar, D.M.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In: Henderson, P. (ed.) Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, pp. 157–167. ACM Press, New York (1984)
13. 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, pp. 235–246. ACM Press, New York (1984)
14. Domani, T., Kolodner, E.K., Petrank, E.: A generational on-the-fly garbage collector for Java. In: Proc. of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 274–284 (June 2000)
15. Appel, A.W., Ellis, J.R., Li, K.: Real-time concurrent collection on stock multiprocessors. In: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, pp. 11–20 (June 1988)
16. Nettles, S., O'Toole, J.: Real-time garbage collection. In: Proc. of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 217–226 (June 1993)
17. Henriksson, R.: Scheduling Garbage Collection in Embedded Systems. PhD thesis, Lund Institute of Technology (July 1998)
18. Bacon, D.F., Cheng, P., 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, pp. 81–92 (June 2003)