Hierarchical Real-time Garbage Collection

Filip Pizlo
Purdue University
pizlo@purdue.edu

Antony L. Hosking
Purdue University
hosking@purdue.edu

Jan Vitek
IBM T.J Watson
Purdue University
jvitek@us.ibm.com

Abstract

Memory management is a critical issue for correctness and performance in real-time embedded systems. Recent work on real-time garbage collectors has shown that it is possible to provide guarantees on worst-case pause times and minimum mutator utilization time. This paper presents a new *hierarchical* real-time garbage collection algorithm for mixed-priority and mixed-criticality environments. With hierarchical garbage collection, real-time programmers can partition the heap into a number of *heaplets* and for each partition choose to run a separate collector with a schedule that matches the allocation behavior and footprint of the real-time task using it. This approach lowers worst-case response times of real-time applications by 26%, while almost doubling mutator utilization – all with only minimal changes to the application code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters, run-time environments; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems.

General Terms Languages, Experimentation.

Keywords Real-time systems, Java Memory management.

1. Introduction

Multi-million line systems are being developed in Java for avionics, shipboard computing and simulation. A key attraction of the Real-time Specification for Java (RTSJ) [7] for such systems is that it makes it possible to develop applications that mix hard-, soft-, and non-real-time tasks in the same environment in a memory-safe way. Unfortunately these advantages may come at the expense of predictability of the real-time subsystems. In Java, one important cause of unpredictability is garbage collection which can sometimes interrupt application code for hundreds of milliseconds. Real-time garbage collectors (RTGC) attempt to mitigate this problem by lowering the worst-case bounds on pause times [4, 10, 23, 22].

The challenge in real-time garbage collection is not only to provide predictably small pause times for real-time tasks, but, just as importantly, to ensure some minimum mutator utilization (MMU) [4]. For any time interval, MMU measures the fraction of that interval guaranteed to be available to the mutator threads. In a real-time system this bounds how much useful work a task can hope to accomplish while the collector is idle. Consider a real-time task scheduled periodically every 10 ms. If RTGC guarantees 50% mutator utilization, the task can count on at least 5ms per period (excluding any additional overhead imposed by RTGC for read/write barriers, allocation costs, etc.). In Metronome [4], a state-of-the-art production RTGC algorithm, the lower bound on mutator utilization depends on the maximum allocation rate of all threads running in the virtual machine. Thus, MMU is obtained as a function of the maximum number of bytes allocated in a given time window over all concurrent threads. As MMU is a crucial input to schedulability analysis, avoiding conflating real-time and non-real-time tasks into a single global maximum allocation rate is desirable, otherwise real-time tasks are forced to depend on the behavior of plain Java code.

This paper introduces a new *hierarchical real-time garbage collector* (HRTGC), which allows differentiation between real-time and non-real-time workloads running within a single Java virtual machine (JVM). HRTGC increases the MMU of the real-time tasks without overly impacting the plain Java parts of the system by splitting the heap into a number of disjoint *heaplets* and running a different collector in each heaplet. Each collector can be tuned independently to best match the characteristics (allocation behavior and footprint) of the tasks running in that heaplet. Moreover, the rate at which a collector must run is determined by the maximum allocation rate of the threads running within its heaplet and not all threads in the system. Fig. 1 illustrates HRTGC. In this example, the heap is split into three heaplets. Each heaplet has one real-time collector thread and a number of mutator threads. In a typical configuration of HRTGC, programmers will assign high-priority real-time threads to the leaves of the heaplet tree – since they require less

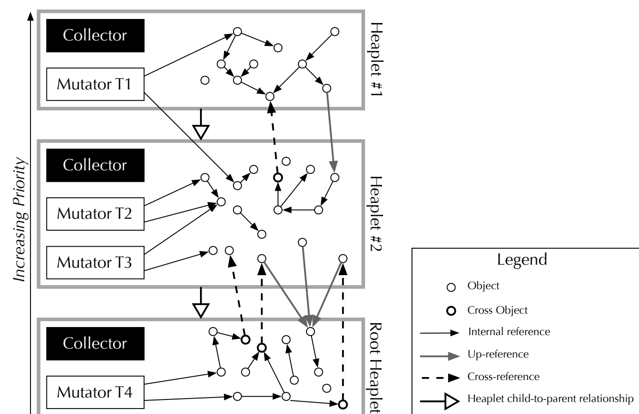


Figure 1. ***Heap hierarchy with heaplets and inter-heaplet references.*** *Cross-heaplet references are allowed to exist freely, giving programs written with HRTGC the same expressive power as those using RTGC. However, the highest-priority thread, T1, is not affected by the collectors associated with the lower-priority heaplets.*

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'07 June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-632-5/07/0006…$5.00.

work on the part of the collector – and each collector thread usually will have higher priority than the threads running in its heaplet. It is noteworthy that with HRTGC, unlike RTSJ, all patterns of references between objects are allowed, though references that cross heaplet boundaries may require additional book-keeping. HRTGC is designed so that references from a child heaplet to an ancestor heaplet cost nothing, a necessary feature since we don't want to penalize access to static variables, while all other cross-heaplet references incur some overhead. It is worth emphasizing differences from the Real-time Specification for Java. In terms of functional correctness, adding a heaplet can never cause the program to fail, whereas adding a new RTSJ memory scope may cause exceptions to be raised by the JVM. The impact of heaplets shows in the performance, footprint, and processor utilization of the mutators.

This paper makes the following contributions:

- **Hierarchical RTGC:** We designed and implemented a prototype hierarchical real-time garbage collector in a high-performance real-time JVM. To the best of our knowledge this is the first partitioned collector to provide real-time guarantees.
- **Predictability:** We target hard real-time systems where predictability is paramount. The algorithm strives to eliminate variability in all collector operations. In particular, barriers and allocation always have predictable performance behavior – as opposed to most other collectors which have "fast" and "slow" paths. Our results demonstrate that even with this restriction performance is competitive.
- **Usability:** The HRTGC API is much simpler than the region-based memory model of RTSJ and much less error-prone. We have refactored RTSJ code to use HRTGC and found that it improved readability and reduced failures.
- **Empirical Evaluation:** To the best of our knowledge we are the first to report performance results for real-time garbage collection that are not based on micro-benchmarks or non-real-time workloads. We have evaluated our implementation of HRTGC on two real-time applications: the RTZen Object Request Broker [14] and the Collision Detector (comprising 202K and 41K lines-of-code, respectively).
- **Performance:** The performance of HRTGC surpasses that of previously published approaches. On one application we see a 26% improvement in response time over non-hierarchical RTGC, and an almost two-fold increase in MMU for large windows. Such improvements are remarkable, considering that the collision detector is not memory-intensive, which means GC-related improvements would not be felt so strongly.

In summary, the proposed HRTGC API and implementation is a simple yet effective extension of the Real-time Specification for Java that will make real-time programs more robust and better able to meet their hard real-time guarantees.

2. Programming model

Unlike most other partitioning garbage collectors, HRTGC requires user input to define the heap partitions. Each heaplet partition represents an independent GC domain that can be scheduled and collected separately, and preempted as necessary based on the priority of its GC thread. This permits establishing tight bounds on the overheads of GC for threads running in different heaplets, subject to per-heaplet scheduling parameters. Partitioning into heaplets and establishing the parent-child relation between heaplets is explicit in the code, however creating cross-heaplet references remains transparent avoiding the need for invasive refactoring of libraries. While beneficial in terms of GC overhead and scheduling, partitioning has overheads. Each independent heaplet collector must actively discover, or be provided, all incoming references from ancestor heaplets. Tracking such cross-references is one source of additional overhead for mutators. To keep these overheads low, programmers must be careful how they allocate objects in heaplets so as to minimize the number of incoming references from ancestor or sibling heaplets.

For developers familiar with the Real-time Specification for Java (RTSJ), this partitioning comes naturally. As in RTSJ, programmers must define a tree of memory regions and decide in which region to allocate. While partitioning a real-time system into mostly disjoint subsystems is arguably good from a software engineering viewpoint – reducing coupling decreases the potential for interference between real-time tasks and enables local reasoning about the different tasks in the application – RTSJ may be too rigid in its enforcement. Any reference from a parent region to one of its children, or between sibling regions immediately leads to a runtime memory access exception. In previous work we have shown the intricate and demanding programming idioms required to avoid errors [20, 2] and reported on anecdotal evidence that RTSJ-style memory management is an important source of software defects. The key difference between RTSJ and heaplets in HRTGC is that cross-heaplet references are permitted, though they may incur overheads. Thus, the only impact of cross-heaplet references may be to degrade the benefits of partitioning but not to endanger functional correctness. Nevertheless, the degenerate case where most reference are cross-heaplet references will likely lead to violations of the non-functional (time and/or space) requirements of a hard real-time system. While more empirical experience is needed, we do believe this is a less severe impediment than the current issues with RTSJ scoped memory.

The HRTGC API is simple, it appears in Fig. 2. The `Heaplet` class reifies the notion of an allocation area. Instantiation of a heaplet reserves space of the specified `size` from which to allocate for that heaplet, and starts a new collector thread for that heaplet. The collector runs with a `priority` and a predefined `schedule`. A heaplet has an `enter()` method that takes a `Runnable` and executes it with allocation directed to that heaplet. Additionally, to allow the programmer to co-locate objects within a heaplet, the `heapletOf` static method returns the allocation heaplet of a given object. Heaplets are multithreaded and re-entrant.

```
public class Heaplet {
    public Heaplet(Heaplet parent, int size,
                   String schedule, int priority);
    public void enter(Runnable logic);
    public static Heaplet heapletOf(Object o);
}
```

Figure 2. HRTGC API. *A `Heaplet` represents a partition of the heap. The heaplet hierarchy is set up implicitly at creation, a new heaplet becomes a child of the current heaplet.*

There is a single distinguished *root heaplet* to which all allocation is initially directed. Programmers define heaplets as hints to refine memory management scheduling policies. Other than their impact on scheduling, heaplets dictate neither lifetime nor thread-locality for their allocated objects. Modest changes to the JVM's own use of allocation are required to support heaplets. Monitors are always allocated in the same heaplet as the object they serve. Static initialization is always performed in the root heaplet. Additionally, the current heaplet is inherited across threads: if a thread is instantiated in a given heaplet, it will run in that heaplet by default. Finally, though not supported in our current implementation, finalization can be performed in any heaplet.

2.1 Refactoring Existing Real-time Java code

How does one refactor an existing RTSJ application to use a hierarchical collector? Our experience suggests that the modifications needed are minimal. A correct RTSJ program is guaranteed to have no costly cross-heaplet references (it may have upward references but these are free with HRTGC), it is thus an ideal candidate for HRTGC as its developers have already architected the code so that objects accessed by real-time threads are segregated from the main Java heap. Thus, turning an RTSJ application into an HRTGC one typically requires changing a few API calls. RTSJ programs use the `ScopedMemory` API to manage allocation of objects that should not be touched by the garbage collector. The virtual machine enforces a separation, via run-time checks, between objects allocated in scoped memory and standard Java objects. Any reference from a heap-allocated object to scope-allocated object will cause an exception to be thrown by the JVM. A hierarchical collector is more permissive – such references are allowed at a small performance overhead.

In RTSJ programs, using `ScopedMemory` is often tricky because programmers have to be very careful to avoid run-time errors. In one of our benchmarks (Zen) we found over 500 lines scattered over many files that were dealing with memory management. Refactoring the application to use HRTGC meant deleting all of this code (or rather linking against a dummy library that ignores calls to `ScopedMemory`), and adding the following to the application:

```
Heaplet area = new Heaplet( 200*KB, "--R", 25);
area.enter(new Runnable(){
      public void run() { ZenDemo.main(); } });
```

This code creates a new `Heaplet` with 200KB of memory. The schedule argument, `"--R"`, means that the collector runs every third time quantum (these have a default value of 1 ms) at priority 25. Then the `enter()` method is invoked with a freshly created `Runnable` to be executed within the allocation context of the heaplet. Interestingly, the original program had several different memory regions set up for different real-time tasks. We were surprised to find that a single heaplet was sufficient to meet the real-time requirements of our benchmarks.

In our two example applications we only experiment with two or three heaplets, for Zen and our other benchmark, the Collision Detector, respectively. Clearly there will be cases where a richer heaplet hierarchy is needed, but our experience suggests that the HRTGC version of an application will be simpler than the corresponding RTSJ system.

3. Hierarchical Real-time Garbage Collection

Hierarchical real-time garbage collection relies on a hierarchy of cooperating heaplet collectors running at different rates in separate real-time threads. Here we implement these collectors as a variant of the Metronome [4] RTGC, though other algorithms are possible. The key challenge for HRTGC is defining and implementing the protocols that coordinate the collection activity of the JVM.

In HRTGC, a *heaplet* is a fixed-size partition of the heap with an associated heaplet collector. Except for the distinguished *root heaplet*, every heaplet has a single parent. References between objects are categorized as *internal references* when the source and target objects are located in the same heaplet, *up-references* when the target object is located in an ancestor heaplet, and *cross-references* when the target is any other heaplet. We organize the handling of internal and up-references so that they can be established for free, while establishing a cross-reference comes at a non-trivial price.

HRTGC is based on a simple incremental mark-sweep snapshot-at-the-beginning collector. Fig. 3(a) shows the basic technique

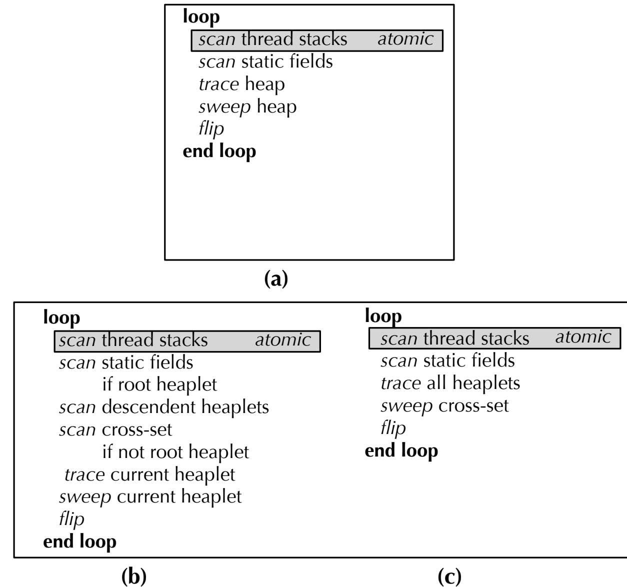


Figure 3. ***Collection algorithms.*** *(a) shows the basic mark-sweep algorithm that we base HRTGC on. The collector runs in a high-priority thread that collects indefinitely. Only the stack scanning is atomic; all other phases are incremental, with the collector periodically yielding to the mutator. In (b) we show the modified algorithm used for heaplet collection, while in (c) we show the cycle collector.*

which, with the exception of thread stack scanning, is fully preemptible. In a real-time setting the basic RTGC algorithm runs in an infinite loop, proactively checking whether it should yield to a mutator thread. Our approach is to use this same algorithm to collect each heaplet; however, it cannot be used for this purpose without modification since it does not account for up-references and cross-references. Fig. 3(b) and (c) give an overview of HRTGC. Each heaplet collector, shown in Fig. 3(b), is augmented to look for up-references by scanning descendent heaplets; cross-references are handled by scanning the *cross set* (see Sec. 3.1). For the root heaplet, static fields must also be processed. Of course, one downside to collecting heaplets independently is the possibility that cycles of references result in cyclic garbage that cannot be collected by any one heaplet collector. Thus, we augment the separate heaplet collectors with a global cycle collector, shown in Fig. 3(c) and explained in Sec. 3.3. Compaction is not currently supported.

The heaplet hierarchy enforces a containment policy regarding which portions of the global heap need to be scanned for collection of a given heaplet. This policy permits up-references to be created with no additional book-keeping. Cross-references are recorded by a write barrier. This permits a descendent heaplet to be collected independently of its ancestors and siblings, without having to scan those ancestors for incoming references.

The remainder of this section is structured as follows. We first describe the cross set. The correctness invariants that we preserve even in the case of interference from the mutator and between collectors follow. The collection algorithms are discussed in Sec. 3.3, and Sec. 3.4 shows the modifications to object structure that are required for HRTGC. Secs. 3.5 and 3.6 talk about the allocation algorithm and the write barrier, respectively. Finally, Sec. 3.7 talks about worst-case performance and scheduling.

3.1 The Cross Set

In addition to maintaining the heaplet hierarchy, an HRTGC must maintain meta-data to facilitate collection. The most significant component is the *cross set*, which allows us to track cross-references. The cross set is a global data structure, shared by all heaplets, which remembers all objects having outgoing cross-references. Objects are added when a write barrier detects creation of a cross-reference. Once an object is added to the set it will remain there until it is reclaimed or removed by the cycle collector. The cross set is maintained as a sparse array with constant time insertion and deletion. Insertion is managed using a free-list that contains a stack of all the indices in the sparse array that are unused. Both the free-list and the sparse array are sized statically. This requires the user to control how many cross objects she is willing to allow. Additionally, the cross set allows for non-blocking scans – adding or removing elements from the cross set while it is being scanned causes no disruptions.

3.2 Correctness Invariants

Interference between the collector and mutator, or between one collector and another collector, can result in the system's correctness invariants being broken. In this section we consider what correctness invariants we need to maintain, and what techniques we employ to ensure those invariants. Because our system is non-moving, the collector does not interfere with the mutator; hence, we restrict our attention to mutator-collector interference, and collector-collector interference. To aid in handling interference from either the mutator or other collectors, every collector in HRTGC maintains the *weak tri-color invariant* [19]. Each collector labels each object *white*, *grey*, or *black*. White objects are candidates for reclamation. White objects are colored grey when the collector identifies them as reachable. Grey objects are colored black when all of the references they contain have been followed by the collector. Unlike objects in a conventional collector, objects in the HRTGC may at any time be subject to simultaneous marking phases by different collectors – at worst, one for each heaplet and one for the cycle collector – thus, we must attach *multiple* colors to each object.

The weak tri-color invariant states that a black object may only refer to a white object if that white object is also reachable from a grey object through a path of zero or more white objects. A collector that maintains the weak tri-color invariant is said to be a *snapshot-at-the-beginning* collector because as soon as the root objects are turned grey at the beginning of collection, any objects reachable from them form a snapshot that is guaranteed to be traced by the collector. This immediately suggests the strategy for handling mutator-collector interference. We add another action to be performed in the write barrier: when a reference is about to be overwritten, we mark (grey) the referenced object both in the cycle collector and in the object's heaplet. Hence, if a reference to an object in the snapshot is broken, that object is immediately marked, ensuring that no part of the snapshot is lost.

Collector-collector interference is a greater challenge. The set of grey objects for one collector may include objects in a foreign heaplet. Consider the scenario in Fig. 4. Here, Collector #1 (which may be either a heaplet collector or the cycle collector) has a white object referenced from a black object. This is legal under the weak tri-color invariant since there is a grey object in Collector #2's heaplet that also references that white object. However, the grey object is about to be deleted by Collector #2 – for example because all references to it will soon be cleared and Collector #2 will observe that it is unreachable. Thus we have a race: if Collector #1 scans the grey object first, everything will be fine. But there may be a problem if Collector #2 gets to it first.

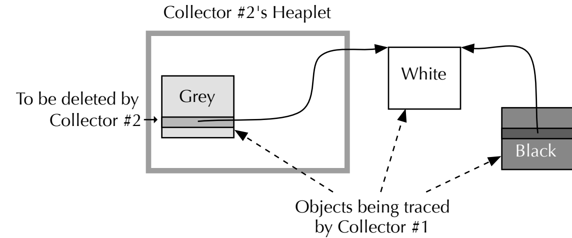


Figure 4. ***A collector-collector interference scenario.*** *The white object is live, being referenced from the black object. Collector #1 counts on being able to find the white object by eventually scanning the grey object. However, Collector #2 is planning on deleting the grey object.*

See Fig. 5 for an illustration of this race. In Fig. 5(a) nothing special needs to be done because the scan completes before the deletion begins. However, in Fig. 5(b) and (c), we need to have some way of ensuring that whatever is referenced from the grey object gets marked, and also that Collector #1 doesn't crash because of an attempt to scan garbage. The solution is to abort either the scan or the deletion, depending on which came second. If, as shown in Fig. 5(b), the deletion begins before the scan begins, we simply abort the scan. To catch all of the references that would be lost, we augment the deletion with a *deletion barrier* that, for each collector that had the deleted object in its snapshot, marks all objects referenced from the deleted object. Because the deletion barrier does not affect collectors that did not have the deleted object in their snapshot, it does not introduce additional object "drag". In Fig. 5(c) the deletion starts during the scan, so we simply abort the deletion. The collector will once again attempt to delete the object on the next collection cycle.

Aborting the deletion in the case of a concurrent scan is accomplished by using a *hold count* associated with each page. When a page contains an object that is being scanned, its hold count is incremented, only to be decremented when the scan completes. Aborting the scan is handled differently depending on how the object is found. In the case of the descendent heaplet scan, the collector doing the scan is iterating over the current in-use objects. If an object is not in-use when it is encountered by the scan, it is ignored. When a collector is scanning the cross set, it is guaranteed that it will not observe any deleted objects, since object deletion involves removal from the cross set. Finally, the cycle collector may enqueue an object on its worklist only to have that object deleted before it is dequeued. This scenario can be broken down into two cases:

1. *Dequeuing garbage:* The pointer dequeued does not point at a valid object. The heap structure used by the hierarchical garbage collector allows us to identify whether an address refers to the base of an in-use object in constant time. If it does not, the cycle collector ignores the object.
2. *Reallocation:* The object that was enqueued may be deleted and a new object may be allocated in its place. In this case the newly allocated object will already be marked in the cycle collector. Hence it is safe to scan it (since it is a proper object, we aren't scanning garbage); on the other hand, since it is already marked, the act of scanning it won't mark any objects that would not have been otherwise marked.

3.3 Collection Algorithms

The Heaplet collector, shown in Fig. 3(b), is a snapshot-at-the-beginning mark-sweep collector. The collector begins by scanning all thread stacks. This is essential: it allows any thread to refer to

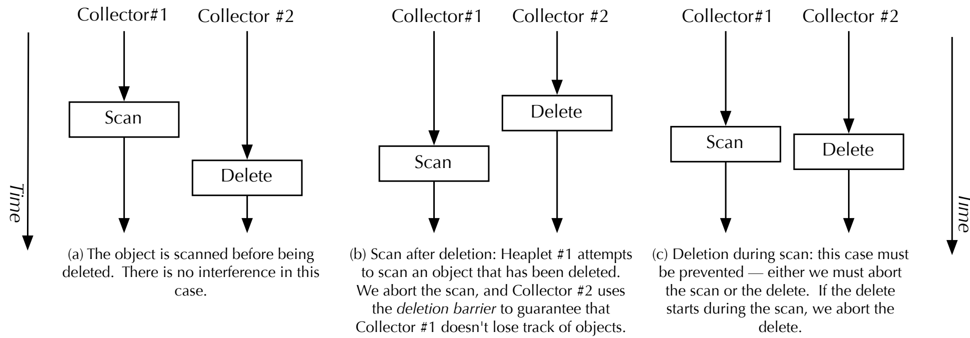


Figure 5. ***The three cases of collector-collector interference.*** *A grey object, as in Fig. 4, is about to be deleted by Collector #2. However, Collector #1 needs to scan it. In (a), the scan occurs before the delete. This is the simple case – no additional coordination is necessary. In (b) the delete occurs before the scan. We handle this with a two-pronged approach: we ensure that Collector #1 can realize that the object was deleted and abort its scan, while at the same time adding a* deletion barrier *to Collector #2 that shades all objects referenced from the object being deleted. In (c) the deletion starts before the scan finishes. In this case we abort the deletion.*

any heaplet without additional book-keeping. We consider static fields virtually part of the root heaplet. Hence, only the root heaplet and the cycle collector scan static fields. If a static field refers to a non-root heaplet, the class object is placed on the cross set. This frees non-root collectors from having to traverse all static fields. Two additional sources of roots may be used by heaplet collectors: descendent heaplets and the cross set. Each collector scans the in-use objects of all of its descendent heaplets. By *scanning* we mean that the collector simply walks the memory occupied by the heaplets, scanning each object that has not been condemned. When designing the algorithm we considered having collectors trace their descendents; however, because every entity that may trace an object would require an additional mark bit and write barrier step, we have decided to use a scan instead because of the lower overheads. Since all collectors are always running anyway, a scan is a good approximation of a trace. Finally, non-root heaplets must scan the cross set to find incoming references from ancestors, siblings, and their descendents.

Following the root scanning phase the collector proceeds with a trace. After this completes, the collector will sweep, deleting any objects that are no longer reachable. Our sweep is eager, thus increasing the predictability of our allocation code and allowing us to assign a simple worst-case bound on allocation. Sweeping is modified to perform three additional actions on object deletion: (*i*) the *deletion barrier* marks objects in foreign heaplets referenced from the object being deleted; all referenced objects are also marked for the cycle collector; (*ii*) if the object is on the cross set it is removed (see Sec. 3.4 for the mechanism that facilitates constant-time cross set removal); and finally (*iii*) if the object's page has a non-zero hold count, the deletion is aborted, and the object will not be deleted until the next collection cycle.

The cycle collector, shown in Fig. 3(c) uses a mark-sweep algorithm similarly to the heaplet collector. It starts by scanning thread stacks and static fields. Then it traces the entire heap, ignoring heaplet boundaries. Finally, it sweeps the cross set. The cycle collector's sweep phase does not do any space reclamation. Instead, it breaks cycles by removing cross objects that it knows to be dead from the cross set. This facilitates space reclamation to happen on the next iteration of the affected heaplet collectors.

3.4 Object Structure

HRTGC requires four additional bits in each object header. To quickly determine if an object is already a cross object, we need a *cross bit*. We need two mark bits – one for the heaplet collector that corresponds to the object and one for the cycle collector. We also need an in-use bit for determining if the object is in use versus being on a free-list. See Fig. 6 for the object header our system uses. Notably missing from the object structure is any reference to the heaplet that owns the object. Instead, we mandate that each page is mapped to only one heaplet, allowing for a fast page-table-based heaplet lookup.

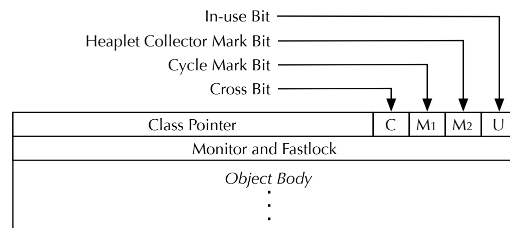


Figure 6. *Structure of the object header in HRTGC.*

Any object may end up on a cross set. If the object is deleted by its heaplet collector, the collector must be able to remove it from the cross set in constant time. For this we need a back pointer from the object to the cross set. The object's monitor is used to store this pointer. Monitors are used in Java for synchronization, and are typically implemented as a separate object referenced from the object they serve. Because most objects do not require a monitor, storing it separately from the object header saves space. We extend the monitor, giving it an additional field, which we call the *monitor token*, that we use to refer back to the cross set entry. Hence, the use of this back-pointer does not inflate heap usage except in the case of objects that already have a monitor (which is rare) and objects that end up on the cross set (which will be rare in the case of a well-chosen heaplet hierarchy).

Fig. 7 shows the cross set in use. In this example, an object in heaplet #1 wishes to point at an object in heaplet #2. The source object is placed on the cross set, and to facilitate constant-time removal, we inflate the monitor and install a back-pointer in the monitor token.

The monitor needs to be available when an object is being freed. If the object is being freed then neither the object nor its monitor are reachable. Hence it is possible that the monitor gets freed first, if it comes first in memory, leading to memory corruption when the collector accesses the monitor after it has been freed. To prevent

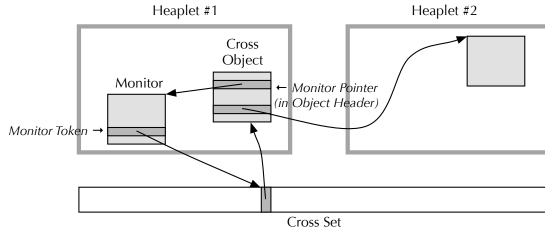


Figure 7. ***Example of the cross set and monitor token in use.*** *An object in heaplet #1 wishes to point at an object in heaplet #2. Because heaplet #2 is not an ancestor of heaplet #1, we must place a reference to the object on the cross set. To be able to remove the object from the cross set in the future, we inflate the object's monitor and store a back-pointer in the monitor token.*

this, we pin the monitor (not in the sense of preventing it from moving, but in the sense of preventing it from being collected even if it is not reachable), until its object is freed.

3.5 Allocation

Allocation in HRTGC requires checking which heaplet the thread is currently allocating from. We maintain a thread-local variable that refers to the current heaplet, as most recently designated by `Heaplet.enter()`. The allocation routine dispatches allocation requests to that heaplet, which chooses how to handle the request.

Following allocation we initialize the new object's header so that its class pointer refers to the correct class and the four GC bits are set appropriately (see Fig. 6 for the object header structure). All collectors in HRTGC allocate black (i.e., as if the collector had already marked them live), so we set both of the mark bits (M_1 for the cycle collector and M_2 for the heaplet collector) to true. Then we set the used bit (U) to true and the cross bit (C) to false.

3.6 Write Barrier

We use a write barrier to detect when an object needs to be placed on the cross set, and also to maintain the weak tri-color invariant for both the cycle collector and the heaplet collectors. To maintain the weak tri-color invariant, we shade the old referent on write. Hence the write barrier has three tasks to perform:

1. Place objects on the cross set when a cross reference is established.
2. Perform the shade-old-referent-on-write action for the cycle collector.
3. Perform the shade-old-referent-on-write action for the collector that corresponds to the old referent.

Fig. 8 shows the write barrier mechanism. The `addCross` auxiliary function is called if there is a need to add the object to the cross set. This function manages the cross set, cross bit, and the monitor token to ensure proper bookkeeping.

It should be noted that like [4] and [10], we do not support truly parallel thread execution – our implementation is restricted to a uniprocessor setting. Thus, we do not have to worry about concurrency in the write barriers; instead we simply disable thread switching during the write barrier.

3.7 Worst-case Performance

The HRTGC is a hard real-time collector designed to provide strong worst-case performance guarantees. As in [4], the user is allowed to specify the collector schedule *a priori*. Further, we make critical collector functions called by the mutator (such as allocation, barriers, and heaplet entry) highly predictable. In this section we lay out what guarantees are provided by HRTGC.

```
void writeBarrier(word_t *referee, int off, word_t *newObj) {
   word_t *ptr = referee + off, *oldObj = *ptr;
   word_t headerword = *oldObj;
   if (!headerword.M1) {
      headerword.M1 = true;
      cycleCollector.enqueue(oldObj);
   }
   if (!headerword.M2) {
      headerword.M2 = true;
      Heaplet *oldObjHeaplet = heapletMap[oldObj / pagesize];
      oldObjHeaplet->enqueue(oldObj);
   }
   *oldObj = headerword;
   Heaplet *newObjHeaplet = heapletMap[newObj / pagesize];
   Heaplet *refHeaplet = heapletMap[referee / pagesize];
   if (newObjHeaplet != refHeaplet &&
      !(newObjHeaplet isdescendentof refHeaplet))
      addCross(referee);
}
```

Figure 8. ***HRTGC write barrier.*** *First we shade the old referent in both the cycle collector and the heaplet collector. The M_1 mark bit is used for the cycle collector, while the M_2 mark bit is used for the heaplet collector. After shading the old referent, we check if the store would cause the referee to become a cross object. If so, we call the `addCross()` function, which adds the object to the cross set.*

Priority-preemptive scheduling. HRTGC assumes a priority-preemptive scheduler. We permit the user to specify a priority for each heaplet collector, including the root heaplet collector and the global cycle collector. HRTGC allows the priority of a heaplet collector to be either lower or higher than the priorities of threads using the heaplet; thus HRTGC can be used to emulate either the Henriksson [10] scheduling style or the style of Metronome [4].

Collector schedules. Heaplet collector scheduling is a direct extension of [4]. Whereas Metronome only has one collector, we have many: a collector for each heaplet, plus the cycle collector. To facilitate scheduling of *multiple* collectors, we allow the user to specify not only how much time is given to each collector, but precisely when each collector will be released. Because the underlying scheduling mechanism of a typical RTOS and RTJVM is quantized – based on *scheduler quanta* driven by a hardware clock interrupt – the HRTGC's API for determining collector schedules is also quantized. For each collector the user supplies a finite scheduling sequence S as follows:

$$S = Q^+$$
$$Q = \text{`R'} \mid \text{`-'}$$

where Q corresponds to a quantum, and the value '`R`' indicates that the collector should run, and '`-`' indicates that it should yield. The scheduler steps to the next entry in S at the beginning of each new quantum; when the end of the sequence is reached, it wraps around to the beginning. For example, a schedule that gives a 50/50 mutator/collector split would be specified as either '`-R`' or '`R-`'. But the strength of this mechanism is that it allows the user to schedule multiple collectors. For example, in a two collector setting the user may use '`--R-`' for one collector and '`R---`' for the other, ensuring that the two collectors never attempt to run at the same time.

Allocator and Barrier performance. Both the allocator and the write barrier are designed to be highly predictable. The allocator's worst-case performance is nearly identical to the best-case. Since there is no "slow path" that would lead to performance outliers in the allocator, it is sensible to empirically determine the worst-case performance of any procedure that uses allocation. A

slightly weaker claim can be made for the write barrier. The barrier has two performance modes: one for establishing internal and up-references, and another for cross-references. For internal and up-references the barrier is designed to run at worst-case when the collector is not active. Thus, to determine the worst-case performance all one needs to do is disable the collector. Paradoxically, when the collector activates, the performance of the barrier will actually *improve*. This means that when accounting the worst-case degradation to throughput due to the collector activating, the barrier does not need to be taken into account.

However, the barrier has a different performance mode for cross-references. Establishing cross-references is expensive, though predictably so (the performance is still constant time, and does not depend on either page size or object size). Thus, to avoid an overly conservative estimation of the worst-case performance of a procedure, the programmer must be able to identify those heap stores that may generate cross references. To aid the programmer in this task, HRTGC includes a cross reference profiler that will identify the types that are involved in cross reference creation. Thus, in practice we have not found it to be difficult to identify all cross reference creation sites.

Collector performance. Each collector's running time is trivially bounded by the total size of the entire heap. But some heaplets can be shown to require significantly less work. The amount of work $W(h)$ necessary to collect a heaplet h depends on that heaplet's size $S(h)$, the objects on the cross set (we say that $S(C)$ is the total size of all objects on the cross set), and the total size of that heaplet's descendents in the hierarchy (we write $g <: h$ to mean that g is a descendent heaplet of h). We can bound $W(h)$ as follows:

$$W(h) \quad is \quad O\left(S(h) + S(C) + \sum_{\forall g <: h} S(g)\right)$$

Thus, a heaplet with no descendents can be collected in the time it takes to trace the heaplet itself, and scan all of the objects in the cross set. The cross set is bounded in size (see Sec. 3.1), so the user has the ability to set tight bounds on heaplet collector performance. A typical use of HRTGC thus only assigns high collector priorities to small *leaf* heaplets, which have no descendents. Real-time tasks are then restricted to allocating in small leaf heaplets, and are prioritized to preempt larger heaplet collectors. This is the mechanism by which HRTGC can be used to achieve higher mutator utilization and faster response times than a conventional RTGC.

On the other hand, HRTGC is not designed to yield good performance if child heaplets are too large, if the hierarchy is too deep, or if the cross set is too large. We have not found this to be a problem in any of our benchmarks.

4. Evaluation

The goal of HRTGC is to reduce the response times and increase the mutator utilization of real-time tasks provided that two or more heaplets are used. We conducted a number of experiments to evaluate the extent to which HRTGC improves response times and MMU for real-time benchmarks. We also construct a microbenchmark to further characterize the benefits of HRTGC. Finally we measure the raw throughput overheads of HRTGC's more complicated collection, allocation, and write barrier mechanisms by running the SPECjvm98 benchmark suite without any heaplet instrumentation.

These experiments were performed on a 3.8Ghz Pentium 4, with 4GB of physical memory. The operating system used was Linux (Ubuntu kernel version 2.6.15-25-686); all benchmarks were run with the `SCHED_FIFO` scheduling policy. The only threads running at a higher priority were OS kernel threads. We implemented HRTGC and a baseline RTGC similar to the Metronome by modifying the Ovm Real-Time JVM developed by researchers at Purdue. The Ovm RTSJVM is a reasonable vehicle for our work as it features a high-performance ahead-of-time compiler and is the first RTSJVM to have been deployed and flight tested [3].

The main thrust of our evaluation is using two freely-available real-time benchmarks: the Collision Detector (CD) and RTZen. *In both cases we add 227_mtrt from the SPECjvm98 benchmark suite as a noise-maker because of its high allocation rate and large footprint.*

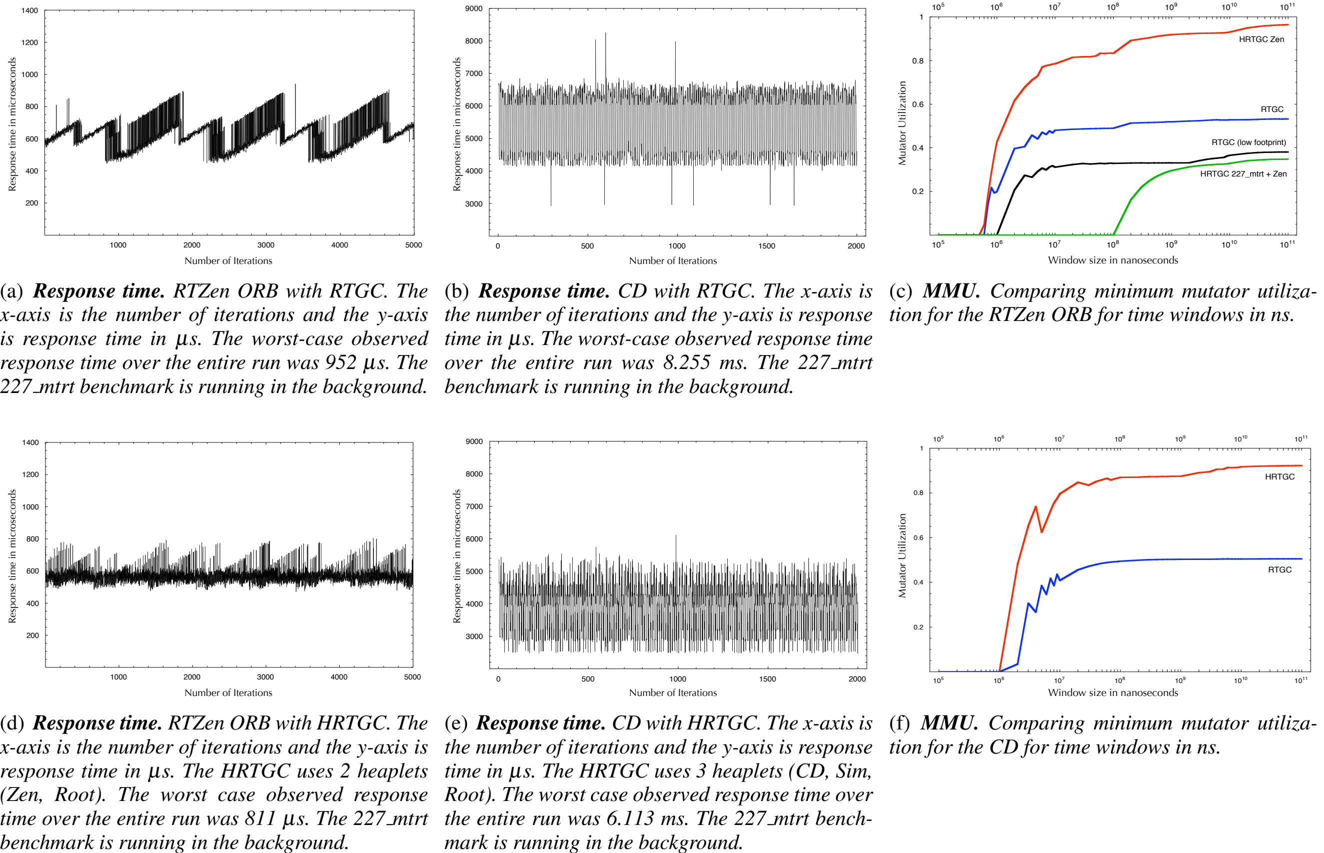
CD is a 41Kloc RTSJ program with two real-time threads. One thread is a periodic hard real-time thread that detects collisions in data generated by a simulator thread. The input is a complex simulation involving 266 aircraft. Our experiment is set up as follows. The real-time CD thread is mapped to one heaplet, the simulator is mapped to a sibling heaplet. The rest of the system, including two threads running the 227_mtrt benchmark, is in the root heaplet. The simulator thread communicates with the detector by copying data allocated in the simulator's heaplet into the detector's heaplet. When run in RTGC, the heaplet API calls are removed. *Response time* is measured as the time between the reception of a frame in the CD thread and the end of the processing of that frame. What we want to show is that using HRTGC allows us to reduce worst-case response times while maximizing the mutator utilization of the CD thread.

RTZen is a 202Kloc multithreaded real-time CORBA ORB developed at U.C. Irvine [14]. We measure RTZen 1.1 running the DOOM demo application (bundled with the RTZen release) with one client. Client and server are on the same machine. The client uses the RTSJ scoped memory API and thus does not experience GC pauses. The server is configured by running the `main()` method of the DOOM application in a heaplet. The static variables and VM data structures are in the root heaplet. Two threads running the 227_mtrt benchmark are also in the root heaplet. In the case of RTGC we simply use the original Zen code, without the modified `main()` method. Response time measures the processing of one request, which includes marshalling on the client, local socket I/O, marshalling on the server and server side processing. What we want to show is that the response times under HRTGC are better than under RTGC, and that mutator utilization for all of Zen improves. Note that other than instrumenting the `main()` method, no changes to the Zen code base were required in this experiment. Hence, a performance improvement in this benchmark demonstrates not only the potential speed-ups, but also the ease with which they can be attained.

4.1 Response Time

Zen exhibits a 15% improvement in worst-case response times when run in HRTGC, versus using RTGC. In the worst case for RTGC, Zen has a response time of 952μs, while HRTGC sees a worst case of 811μs. Both numbers are the worst case over a random ten-minute run involving 20,000 samples. The improvement in HRTGC is due to the handling of background tasks: in RTGC, we were forced to pick a schedule that results in lower mutator utilization to compensate for 227_mtrt's high allocation rate, while in the HRTGC case we only had to worry about Zen's allocations – the root heaplet collector, which served the 227_mtrt benchmark, ran at a lower priority than Zen, so its activities had no effect on the Zen response times. The Zen heaplet collector ran with a schedule where the collector is running only one sixth of the time.

The pattern of response time behavior can be seen in Fig. 9(a) for the RTGC and in Fig. 9(d) for the HRTGC. We only show a 5K-sample selection to make the structure easier to see. The saw-tooth pattern is caused by the collector's periodicity being slightly out of phase with the Zen client's periodicity – as time progresses, the



(a) ***Response time.*** *RTZen ORB with RTGC. The x-axis is the number of iterations and the y-axis is response time in µs. The worst-case observed response time over the entire run was 952 µs. The 227_mtrt benchmark is running in the background.*

(b) ***Response time.*** *CD with RTGC. The x-axis is the number of iterations and the y-axis is response time in µs. The worst-case observed response time over the entire run was 8.255 ms. The 227_mtrt benchmark is running in the background.*

(c) ***MMU.*** *Comparing minimum mutator utilization for the RTZen ORB for time windows in ns.*

(d) ***Response time.*** *RTZen ORB with HRTGC. The x-axis is the number of iterations and the y-axis is response time in µs. The HRTGC uses 2 heaplets (Zen, Root). The worst case observed response time over the entire run was 811 µs. The 227_mtrt benchmark is running in the background.*

(e) ***Response time.*** *CD with HRTGC. The x-axis is the number of iterations and the y-axis is response time in µs. The HRTGC uses 3 heaplets (CD, Sim, Root). The worst case observed response time over the entire run was 6.113 ms. The 227_mtrt benchmark is running in the background.*

(f) ***MMU.*** *Comparing minimum mutator utilization for the CD for time windows in ns.*

Figure 9. *Performance summary of Zen and CD benchmarks.*

Zen client's requests come increasingly earlier in a GC quantum, leading to increasingly higher round-trip times. The drop-off occurs when the requests start to arrive outside of a GC quantum.

CD shows a more pronounced difference. The worst observed response time for the CD thread with HRTGC is 6.113 ms versus 8.255 ms for RTGC. This is a 26% reduction in response time thanks to HRTGC. It is due to two factors: 1) the CD has a lower allocation rate than Zen, leading to an optimal schedule where the collector ran only for one eighth of the time (as opposed to one sixth), and 2) the CD response times were on average much longer, which places the CD activity window higher on the MMU plot, leading to a more pronounced difference between HRTGC and RTGC. Fig. 9(b) shows the response time pattern for RTGC, while Fig. 9(e) shows the pattern for HRTGC. The saw-tooth pattern is missing because the response time for CD is not observed by an outside source – since the beginning and end of the observations occur when the collector is not running, the amount of time stolen by the collector is quantized. For the RTGC we see four strata: the best case response times occur when the detector gets lucky and experiences no GC interruptions. Above that are two strata corresponding to one or two GC interruptions. Finally the worst case times occur when there are three interruptions. In the case of HRTGC, the detector sees at most one collector interruption.

4.2 Minimum Mutator Utilization

Cheng and Blelloch have defined the minimum mutator utilization (MMU) [8] for a given time interval δ as the minimum CPU utilization by the mutator over all intervals of length δ.

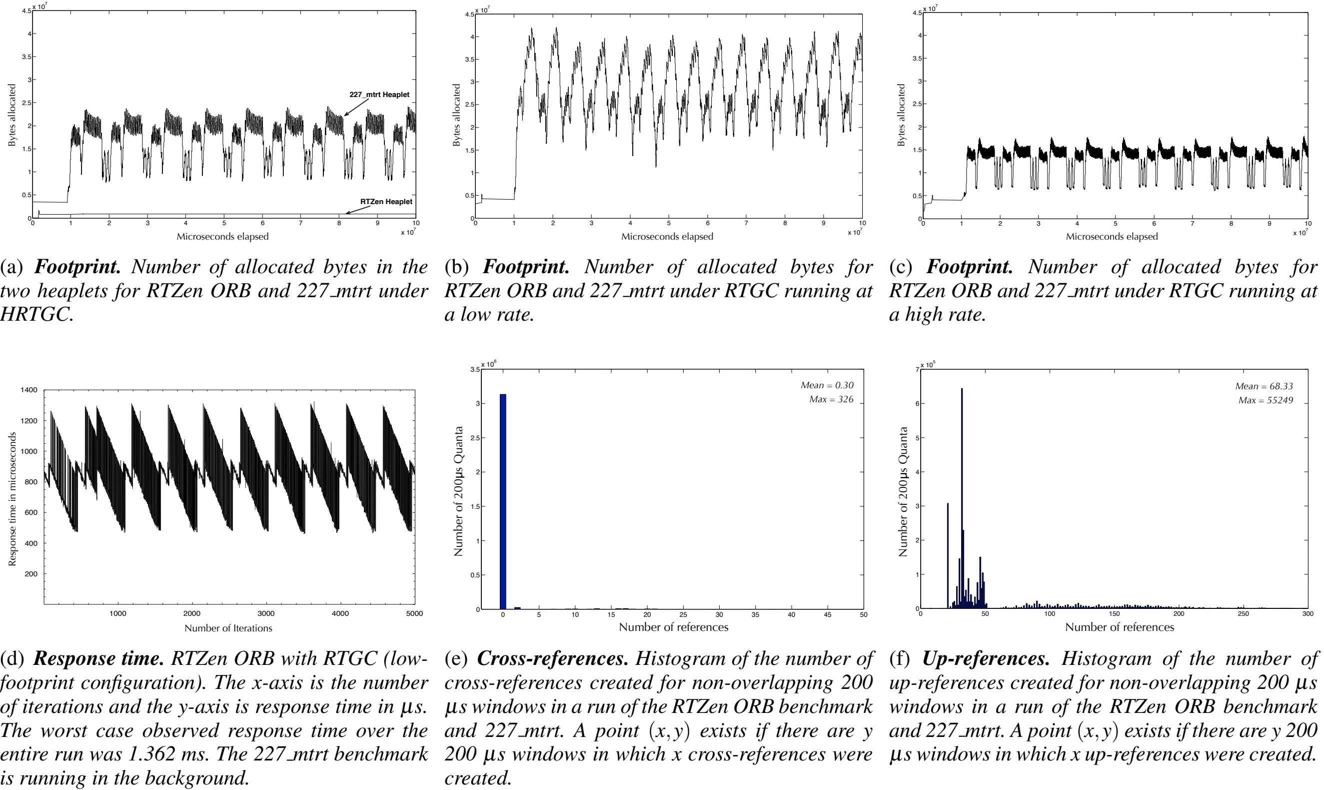
In the case of HRTGC, the MMU must take the different collectors into account. Rather than giving a single MMU for all threads (real-time and plain), HRTGC allows us to differentiate between threads running under different collectors. For each collector priority p we record a separate MMU curve, showing the times when mutator threads of priority $< p$ are not prevented from using the CPU by collector threads of priority $\geq p$.

The Zen MMU, Fig. 9(c), shows the MMU of HRTGC Zen collector, which is the highest priority collector, and the MMU for the union of the Zen and 227_mtrt collectors. We show two RTGC MMU curves, one for the RTGC configuration that we've discussed so far and one for a low-footprint configuration of RTGC (see Sec. 4.3). The graphs show that the real-time task gets significantly higher utilization under HRTGC by penalizing lower priority tasks – in this case 227_mtrt clearly gets worse utilization. The minimum pause time (smallest window) is on the order of 600 μs. The maximum utilization for HRTGC is 97% in Zen and 35% for the mtrt threads. The maximum utilization for RTGC is 53%.

CD has similar results (see Fig. 9(f)). Both collectors experience worst-case pause times around 1 ms, but as the window size increases HRTGC displays much better utilization, with a maximum of 92% utilization, while RTGC peaks at only 51%.

4.3 Footprint

We examine the footprints of Zen running in HRTGC and RTGC. The amount of used memory in bytes in each heaplet of HRTGC is shown in Fig. 10(a), while the total memory usage in RTGC is shown in Fig. 10(b). In the worst case, HRTGC uses a total of



(a) ***Footprint.*** *Number of allocated bytes in the two heaplets for RTZen ORB and 227_mtrt under HRTGC.*

(b) ***Footprint.*** *Number of allocated bytes for RTZen ORB and 227_mtrt under RTGC running at a low rate.*

(c) ***Footprint.*** *Number of allocated bytes for RTZen ORB and 227_mtrt under RTGC running at a high rate.*

(d) ***Response time.*** *RTZen ORB with RTGC (low-footprint configuration). The x-axis is the number of iterations and the y-axis is response time in μs. The worst case observed response time over the entire run was 1.362 ms. The 227_mtrt benchmark is running in the background.*

(e) ***Cross-references.*** *Histogram of the number of cross-references created for non-overlapping 200 μs windows in a run of the RTZen ORB benchmark and 227_mtrt. A point (x,y) exists if there are y 200 μs windows in which x cross-references were created.*

(f) ***Up-references.*** *Histogram of the number of up-references created for non-overlapping 200 μs windows in a run of the RTZen ORB benchmark and 227_mtrt. A point (x,y) exists if there are y 200 μs windows in which x up-references were created.*

Figure 10. Detailed measurements of Zen benchmark performance.

26.4MB, while RTGC uses 43.5MB – so RTGC is 65% worse. The higher memory usage in RTGC is due to the scheduling: in HRTGC, we can afford to run the root heaplet collector at a very high rate, keeping 227_mtrt's memory usage at bay. But to attain competitive response times in RTGC, we must loosen the schedule, resulting in more memory being allocated by 227_mtrt during the collection cycle – hence the larger memory usage. To confirm this we reran the Zen benchmark with RTGC configured to run two thirds of the time. We say that this is a *low footprint* configuration because it improves memory usage by collecting more frequently, at the cost of worsened response times. The memory usage of this modified system is shown in Fig. 10(c), while the response times are shown in Fig. 10(d). The worst case response time in the low footprint configuration is 1362μs, which is 43.1% worse than for HRTGC. Further, as shown in Fig. 9(c), the low footprint configuration results in only 38% utilization in the best case. However, the memory usage is 30% better than HRTGC. This is expected: it is possible to trade off response time for memory usage. However, it would not be possible to reduce the response times of RTGC to be better than HRTGC under any configuration that uses the same total memory as HRTGC.

4.4 Cross references

The purpose of organizing heaplets into a hierarchy is to take advantage of the fact that the majority of cross-heaplet stores involve references going in one direction. In Fig. 10(e) and Fig. 10(f) we show histograms of the number of references created in 200μs windows in the Zen benchmark. Recall that in this benchmark the heaplet hierarchy was not optimized, nor was the Zen code itself modified to optimize heaplet usage. Nevertheless, what we see is that up-references are created at a much higher rate than down-hierarchy references. On average only 0.3 down-hierarchy references are created per 200μs window – hence the rate of down-hierarchy reference creation is only 1.5 KHz. On the other hand, up-references are created at a rate of 342 KHz. To understand why, consider that Zen does not get an opportunity to create many objects outside of its heaplet (if any at all – it would only create such objects from static initializers, since everything else runs in the Zen heaplet). Hence, most down-hierarchy references are from static fields into the Zen heaplet, plus some references from VM objects allocated in the root heaplet to other VM objects created on behalf of Zen in the Zen heaplet. CD has an even more pronounced effect – only three down-hierarchy references are ever created. This is because in the case of CD, we optimized our usage of heaplets to reduce cross-hierarchy references.

4.5 TwoHeap Microbenchmark

In both the Zen and CD benchmarks, we showed an improvement in MMU and response times for the highest-priority task when using two or three heaplets. For Zen we also showed that lower-priority tasks get penalized. In the case of SPEC, we did not take advantage of heaplets, and showed the bare overhead of HRTGC. This raises the question: can HRTGC deliver a net gain in performance for all tasks? Intuitively, if two concurrent real-time tasks have vastly different memory usage patterns, having two heaplets where each heaplet is tuned specifically to the requirements of its task should result in an overall boost to mutator utilization.

Consider that a task with a large footprint but a zero allocation rate never has to collect – if nothing is being allocated, then no garbage is being created. Further, consider that collecting for a task

that has a very small footprint but a high allocation rate is similarly not too difficult – the trace will complete quickly and all of the time will be spent in the sweep. But now consider both tasks running together in one heap. The increase in necessary collection work is not additive: to collect the garbage allocated by the high-allocation-rate task, we need to spend time tracing the large static footprint of the task that does not allocate. Hence, if we take these tasks and run them in separate heaplets, we expect to be able to achieve better mutator utilization *overall*, even when the time required by both collectors is taken into account.

For this we create the *TwoHeap* microbenchmark. This code simply creates two heaplets of equal size, allocates a large footprint in one and a very small footprint in the other. It then starts two threads: one that enters the small-footprint heaplet and allocates at the highest rate possible for small objects (it just loops allocating instances of `Object`), while the other one enters the large-footprint heaplet and allocates at a very low rate – high enough to require only occasional collection. We also enable TwoHeap to run with RTGC by simply removing all Heaplet API calls.

We then exhaustively searched the space of all RTGC and HRTGC schedules (in the case of RTGC it was a 1-D search, in the case of HRTGC it was a 2-D search since we had two heaplets that we were equally interested in; in both cases we restricted our granularity to schedules that contained 20 quanta). For RTGC the best schedule had 14 out of 20 quanta going to the collector, while for HRTGC the best schedule had 11 out of 20 quanta going to the collectors – with the high-allocation heaplet getting 10 quanta and the high-footprint heaplet getting 1 quantum. Hence, even though HRTGC is a slower collector, it gave a net performance improvement for the entire application in this case. See Fig. 11 for the MMUs of HRTGC and RTGC for the TwoHeap microbenchmark.

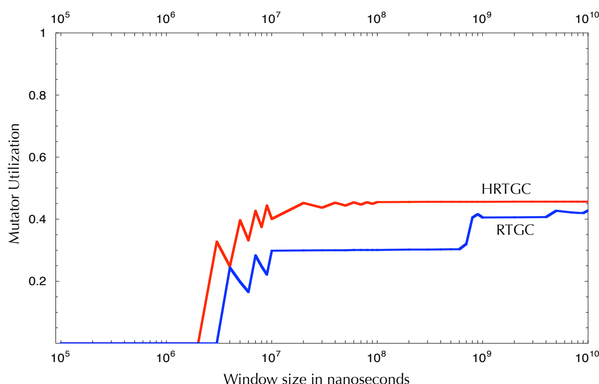


Figure 11. ***MMU.*** *Microbenchmark with different configurations of HRTGC and RTGC.*

4.6 Throughput

The SPECjvm98 benchmark suite is used to evaluate the performance impact of HRTGC on Java applications. In this scenario each benchmark of SPECjvm98 is run within a single heaplet thus not taking advantage of differentiated garbage collection. SPECjvm98 was run with problem size 100. The heap size was capped at 256MB. For each benchmark we record the median of the second iteration of three runs of the benchmark.

Fig. 12 compares the performance of the default mostly-copying garbage collector that comes bundled with the JVM against our real-time collector and the hierarchical real-time collector. The geometric mean overhead of HRTGC over RTGC is 14.6%. This overhead comes in three parts: a heaplet check necessary for allocation, a larger write barrier, and a more complex root scanning and sweeping algorithm in the collector (mainly, the need to run a deletion barrier). In the best case (compress and mpeg) there is no performance penalty and in the worst case (202_jess) we observed a 33.6% slowdown. The mean overhead of RTGC over our mostly-copying collector is 30%.

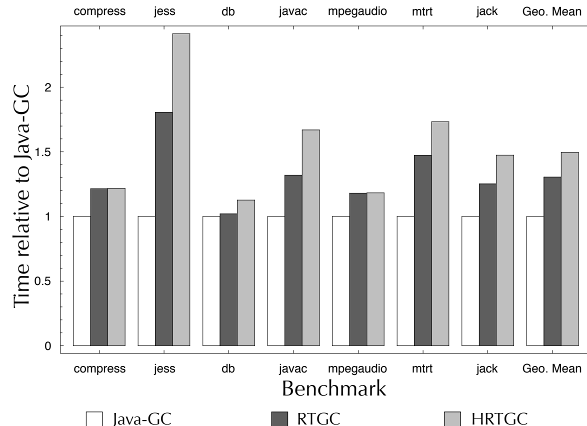


Figure 12. ***Throughput.*** *Comparison of mostly-copying (Java-GC), real-time and hierarchical garbage collectors on the SPECjvm98 benchmark suite. The HRTGC is configured with a single heaplet to emphasize the overheads due to allocation, write barriers and collection.*

5. Related work

Work on real-time collection can be traced back to Baker's incremental copying collector [5]. The central idea behind Baker's collector is decreasing the intrusiveness of a collector by piggy-backing work onto mutator operations. To ensure consistency, a small piece of code, called a read barrier, is inserted by the compiler before every memory read to perform copying, and the allocation code is modified to perform a bounded amount of collection work. The worst-case in a program using Baker's collector involves a copy operation upon every read, and a (large) unit of collection work on every allocation. Hence, even though individual pauses are small, the worst case execution time of an allocation makes Baker's collector unsuitable for hard real-time settings. Put another way, the collector fails to bound its impact on throughput. Baker's collector is said to be *work-based*, in the sense that work done by the mutator leads to work by the collector. Bacon *et al.* [4] investigate different approaches to real-time collection. In Bacon's *time-based* system, the collector interleaves with the mutator at regular intervals. In [10] Henriksson proposes a collector that only becomes active during periods when the real-time tasks are idle. In both collectors, constant time read (or write) barriers are still needed to maintain consistency, and allocation must be made predictable (constant time, or linear in object size). The worst-case bounds on execution time in the mutator become more realistic, allowing the collector to be used in hard real-time systems. In previous work we reported results comparing real-time GC and RTSJ scoped memory [21].

Heap-partitioning garbage collectors have long been used in distributed and persistent environments where the cost of traversing cross-partition references is high [6, 9, 17]. Similarly to our cross list cyclic collector, Juul and Jul [13] used a separate global marking collector to remove unnecessary "inlist" references, regardless of any involvement they may have in a cycle. Other approaches rely on piggy-backing of cyclic collection with partition traces [15]. Yong *et al.* [24] used "remembered sets" to record objects containing inter-partition references that had to be fetched and scanned before tracing a partition. This is similar to our cross objects, except

that their remembered sets were maintained per-partition, whereas our cross set is global. Amsaleg *et al.* [1] used partitioned collection in a transactional database setting, focusing on supporting rollback of transactions, using the database log to process inter-partition references. Other approaches rely on migrating objects among partitions to enable cycle collection [11, 16, 18]. None of these collectors addresses hard real-time systems. Jones and King coined the term heaplet in the context of a compile-time escape analysis for thread-local allocation [12].

6. Conclusion

We have introduced the hierarchical real-time garbage collection algorithm in which the user partitions the heap into heaplets, each of which are independently collected. By partitioning the heap we are able to achieve much better throughput and response times in realistic real-time applications, as compared to our implementation of the Metronome. In particular, we were able to reduce worst-case response times by 26% while almost doubling mutator utilization. These are significant improvements, especially considering that these are not memory-intensive benchmarks and so less susceptible to other improvements in memory management. Adding hierarchical GC permits much more focused processor throughput and utilization by real-time tasks (at the expense of reduced throughput for the non-real-time portions of the application).

Like other RTGCs [4, 10], we do not support multiprocessors, which are a challenge for any real-time system because thread scheduling becomes much less predictable, and it is not possible to ensurex atomicity simply by disabling interrupts and context switches. HRTGC may be more amenable to a highly predictable, low-overhead multiprocessor implementation because heaplets can be used to indicate processor locality. We leave this to future work.

We do not currently support compaction, though there is nothing in our algorithm to prevent compaction, except the challenge to allow cycle collection to track objects even as they are moved without locking. Since none of our benchmarks require compaction, compaction should not change the results presented here.

References

[1] Laurent Amsaleg, Olivier Gruber, and Michael Franklin. Efficient incremental garbage collection for workstation-server database systems. In *Proceedings of the International Conference on Very Large Data Bases*, pages 42–53. Morgan Kaufmann, 1995.

[2] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped Types and Aspects for Real-Time Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 124–147, Nantes, France, July 2006. Springer.

[3] Austin Armbuster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, to appear.

[4] David F. Bacon, Perry Chang, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 285–298, New Orleans, Louisiana, January 2003.

[5] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[6] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1977.

[7] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[8] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 125–136, Snowbird, Utah, June 2001.

[9] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 229–241, Monterey, California, November 1994.

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

[11] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management*, pages 388–403. Springer-Verlag, 1992.

[12] Richard Jones and Andy C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 129–138, Budapest, Hungary, October 2005.

[13] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 103–115, 1995.

[14] Arvind S. Krishna, Douglas C. Schmidt, and Raymond Klefstad. Enhancing Real-Time CORBA via Real-Time Java Features. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 66–73, Hachioji, Tokyo, Japan, March 2004.

[15] Bernard Lang, Christian Queinnec, and José M. Piquer. Garbage collecting the world. In *Conference Record of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 39–50, Albuquerque, New Mexico, January 1992.

[16] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. *Distributed Computing*, 10(2):79–86, 1997.

[17] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In *Proceedings of the ACM International Conference on Management of Data*, pages 313–323, June 1997.

[18] J. Eliot B. Moss, David S. Munro, and Richard L. Hudson. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proceedings of the Seventh International Workshop on Persistent Object Systems*, pages 140–150. Morgan Kaufmann, 1997.

[19] Pekka Pirinen. Barrier techniques for incremental tracing. In *Proceedings of the ACM International Symposium on Memory Management*, pages 20–25. ACM, March 1999.

[20] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC'04)*, Vienna, Austria, May 2004.

[21] Filip Pizlo and Jan Vitek. An empirical evalutation of memory management alternatives for Real-Time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006.

[22] William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 76–85, San Jose, California, October 1994.

[23] Fridtjof Siebert. Real-time garbage collection in multi-threaded systems on a single processor. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 277–278, Phoenix, Arizona, December 1999.

[24] Voon-Fee Yong, Jeffrey F. Naughton, and Jie-Bing Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proceedings of the International Conference on Data Engineering*, pages 120–131. IEEE Computer Society, 1994.