Scheduling Real-Time Garbage Collection on Uniprocessors

TOMAS KALIBERA, University of Kent, Canterbury
FILIP PIZLO, ANTONY L. HOSKING, and JAN VITEK, Purdue University, West Lafayette

8

Managed languages such as Java and C# are increasingly being considered for hard real-time applications because of their productivity and software engineering advantages. Automatic memory management, or garbage collection, is a key enabler for robust, reusable libraries, yet remains a challenge for analysis and implementation of real-time execution environments. This article comprehensively compares leading approaches to hard real-time garbage collection. There are many design decisions involved in selecting a real-time garbage collection algorithm. For time-based garbage collectors on uniprocessors one must choose whether to use *periodic*, *slack-based* or *hybrid* scheduling. A significant impediment to valid experimental comparison of such choices is that commercial implementations use completely different proprietary infrastructures. We present Minuteman, a framework for experimenting with real-time collection algorithms in the context of a high-performance execution environment for real-time Java. We provide the first comparison of the approaches, both experimentally using realistic workloads, and analytically in terms of schedulability.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special Purpose and Application-based Systems—*Real-time and Embedded Systems*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms: Design, Reliability

Additional Key Words and Phrases: Joint scheduling, real-time garbage collection

ACM Reference Format:
Kalibera, T., Pizlo, F., Hosking, A. L., and Vitek, J. 2011. Scheduling real-time garbage collection on uniprocessors. ACM Trans. Comput. Syst. 29, 3, Article 8 (August 2011), 29 pages.
DOI = 10.1145/2003690.2003692 http://doi.acm.org/10.1145/2003690.2003692

1. INTRODUCTION

Managed languages such as Java and C# are increasingly being considered for real-time applications. From both technical and scientific standpoints the most interesting challenge this presents is how to reconcile efficiency and predictability in the memory management subsystem of these languages. To relieve programmers from having to deal with deallocation of data structures and to eradicate memory access errors, managed languages rely on garbage collection for reclaiming unused memory. A number

This work was supported by the National Science Foundation under grants Nos. CCF-0702240, and CCF-0811691, and by Microsoft, Intel, IBM, and the Ministry of Education of the Czech Republic under grant no. MSMOO21620838. Opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.
This research was partially done while A. L. Hosking was on leave as Visiting Fellow, Research School of Computer Science, Australian National University, and T. Kalibera was at Purdue University and Charles University, Czech Republic.
A preliminary version of this work was presented at the 30th IEEE Real-Time Systems Symposium.
Corresponding author's address: T. Kalibera, School of Computing, University of Kent, Canterbury CT2 7NF, United Kingdom; email: t.kalibera@kent.ac.uk.

DOI 10.1145/2003690.2003692 http://doi.acm.org/10.1145/2003690.2003692

of time-triggered real-time garbage collection algorithms have been proposed in the literature and implemented in commercial products. This article attempts to clarify some of the claims that have been made about these algorithms in the scientific and marketing literature. We do this analytically by comparing the schedulability characteristics of the algorithms, and empirically through a careful repetition study in which the algorithms were independently reimplemented in a different environment and compared on real-time workloads and standard benchmarks.

Garbage collection is at heart a simple graph reachability problem, operating over a directed *object graph* comprising objects (nodes) and their references (edges). An application program dynamically allocates heap storage (objects) and manipulates those objects by reference. References may be held by the application at any time in global variables and thread stacks (including registers). The application may also create references between objects by storing references in the heap. From the perspective of the garbage collector (GC), the application program acts as a *mutator* of the object graph. The job of the GC is to determine which objects are *dead*: no longer reachable by the application. Dead objects can safely be reclaimed. The remaining objects are considered to be *live*: accessible at some time in the future by the application. Live objects may also be moved to reduce memory fragmentation.

In a real-time setting it is not practical to collect the heap atomically with respect to the mutator, since stopping the mutator threads results in GC pauses that may cause application threads to miss their deadlines. Thus, a real-time garbage collector (RTGC) must work *incrementally*, interleaved with the normal execution of the real-time tasks. Of course, this means that a real-time GC must cope safely with updates performed by the mutator tasks while it is in the process of reclaiming memory. *Work-based* collectors divide all collection work into fixed-size increments executed by the application at allocation time. *Time-based* collectors, our focus in this work, perform the collection work in a dedicated collector thread. Not surprisingly, there are many research challenges to designing a real-time GC algorithm that is predictable, maximizes throughput, decreases pause times, and keeps memory overheads low.

The context for our work is a Java virtual machine with RTGC and with support for the Real-time Specification for Java (RTSJ) [Bollella et al. 2000], an extension to the Java programming language that is suitable for hard real-time applications as demonstrated by our previous work [Armbruster et al. 2007; Pizlo and Vitek 2008; Honig Spring et al. 2007]. In our RTSJ implementation, a static compiler translates Java code into C ahead of time and then compiles it to machine code using an off-the-shelf C compiler such as GCC. Thus, programmers need not worry about the impact of dynamic loading and just-in-time compilation on the predictability of their programs, and we can focus on GC. We target uni-processors in general as they represent the majority of today's embedded market and, in particular, RTEMS and the LEON architecture used by the European Space Agency for satellite control [Kalibera et al. 2009c].

The contributions of this article are the following.

—*Minuteman.* We have implemented a framework for experimenting with RTGC algorithms that provides support for defragmentation and pluggable scheduling strategies in a high-performance real-time execution environment.
—*Schedulability.* We provide schedulability tests for both periodic and slack scheduling of GC, as well as a hybrid combination of the two. We have compared the scheduling strategies based on simulated workloads.
—*Evaluation.* We empirically evaluate these alternative RTGC algorithms on a number of standard benchmarks as well as on a real-time application. We have demonstrated

the Minimum Mutator Utilization metric that has sometimes been advocated as a way to evaluate real-time collectors is not suitable for measuring slack-based GCs.
—*No Size Fits All.* We find that neither slack nor periodic scheduling is superior to the other. We show that some workloads are only schedulable with periodic GC, while others only with slack GC. The hybrid approach performs best in our experiments.
—*Repeatability.* Our work is the only freely available, open source implementation of the two leading time-based RTGC algorithms.

Minuteman is the first system in which meaningful "apples-to-apples" comparison of different RTGC algorithms can feasibly be made, with results that are not confounded by differences in the environment that are not relevant to GC. A modern GC has a profound impact on aspects of the execution environment ranging from synchronization to compiler optimizations. To evaluate a GC one must account for indirect overheads due to choices such as data layout and code generation. This can only be done in the context of a complete system with representative workloads. When GCs are implemented in different systems, it is almost impossible to compare results, as performance discrepancies may be due to spurious differences. One of our goals was to engineer an experimental platform that is feature-complete and close enough in performance and predictability to production-quality systems that it allows meaningful comparison of different algorithms.

2. REAL-TIME GARBAGE COLLECTION

The goal of a *real-time garbage collector* is to bound space and time overheads of memory management. Since many real-time applications must operate with limited CPU and memory resources, it is essential that the overhead of the GC be small enough to fit in that budget and to enable developers to reason about the impact of a particular GC algorithm on their application. The sources of space overhead for GC are the mark bits used to record reachable objects, the fragmentation resulting from segregated allocation, the heap metadata, and the space reserves. Time overheads come from reference tracing, object scanning, and any object copying performed by the GC, plus the cost of *barrier* operations that may be imposed on the mutator tasks at object allocation, at reads and writes of references in the heap, and on any other heap accesses.

Time predictability is often the main concern when selecting a RTGC. From the point of view of a real-time task that must meet a deadline, three things matter: (a) what is the maximum blocking time due to GC, (b) how many times can it be interrupted by GC, and (c) what is the worst-case slowdown due to the extra barrier checks needed on heap reads and writes? From the point of view of the GC, the question is whether it can keep up with allocation requests and ensure that the system will not run out of memory. One important design dimension in GC algorithms is how to schedule the GC task. The literature on time-triggered RTGC algorithms presents two main alternatives: *slack-based* scheduling as first proposed by Henriksson [1998], and adopted in the Sun Microsystems Java RTS product [Bollella et al. 2005], and the *periodic* scheduling of Metronome [Bacon et al. 2003b] adopted in the IBM Websphere Real-Time product [Auerbach et al. 2007]. Auerbach et al. [2008] later proposed a *hybrid* extension of periodic scheduling, which aims at combining the advantages of both alternatives.

Slack scheduling of GC runs the collector in a separate real-time thread which has a priority *lower* than any other real-time thread. This has the advantage that the GC will never preempt a real-time thread, thus providing a simple and easy to understand answer to points (a) and (b). Complexity arises from the fact that the GC has to be interruptible at any time by a higher-priority real-time thread, and that there

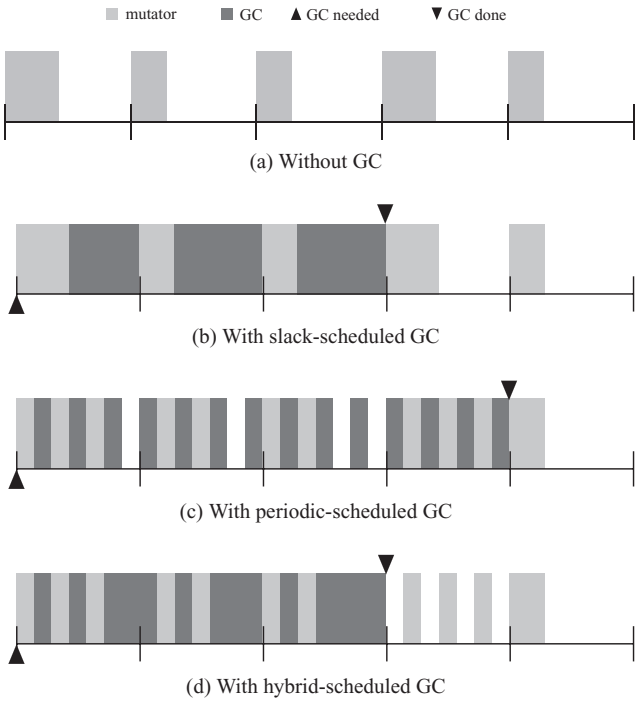


Fig. 1. Sample schedules for a periodic mutator task and different GC scheduling strategies. In Figures (c) and (d) the GC thread preempts the mutator thread. In Figures (b) and (d) the GC thread runs during slack time and can be preempted by the mutator thread.

must be enough "slack" in the schedule to leave time for the GC to satisfy the allocation operations performed by the application. In particular, this means that there must be enough available physical memory to sustain the peak memory needs of any computation done by a periodic real-time thread during its release. Although the GC does not directly interfere with application tasks, there is an indirect cost due to the compiler-inserted barriers needed to make incremental collection possible (point (c) in the previous paragraph).

Periodic scheduling of GC runs the collector thread according to a predefined GC schedule at the *highest* real-time priority. This means that, at regular intervals, GC will preempt application threads and perform a fixed amount of GC work. Since the amount of work performed by GC is fixed and the intervals are known *a priori*, it is possible to come up with an answer to questions (a) and (b). The peak memory requirements that the system must sustain are now bounded by the maximum amount of time the thread can run uninterrupted. As for (c), there are also compiler-inserted barriers because the GC must be incremental.

Hybrid scheduling of GC ensures that the collector thread runs periodically, but allows the collector to use system idle time like slack GC scheduling. This is semantically equivalent to changing the priority of the GC thread from the lowest to the highest in the system. While hybrid scheduling shows that the two approaches are not mutually exclusive, there has been no comparison of the tradeoffs involved in selecting a particular strategy.

The differences among the scheduling approaches are illustrated graphically in Figure 1.

3. THE MINUTEMAN RTGC FRAMEWORK

We have implemented a framework for experimenting with uniprocessor RTGC algorithms called Minuteman. Our framework is built on top of the Ovm real-time Java virtual machine. It is implemented, like the rest of Ovm, in Java, and compiled to C by an ahead-of-time compiler. By slightly restricting the Java reflection API, it is possible to compile the virtual machine statically together with all application code and the needed libraries into a single binary executable for the target platform. A detailed description of Ovm can be found in our previous work [Armbruster et al. 2007]. Ovm implements most of the Real-time Specification for Java [Bollella et al. 2000] and has been deployed on ScanEagle unmanned aerial vehicles in a collaboration with the Boeing Company.

The Minuteman framework supports a wide range of GCs that differ in scheduling, incrementality, defragmentation, predictability of barriers, and the representation of arrays. Scheduling options include slack and periodic, plus hybrid as a combination of both. Incrementality support ranges from non-incremental collection, through selective incrementality of different GC operations, up to full incrementality with incremental stack scanning. Defragmentation can be disabled to obtain a non-moving GC, or it can be enabled with Brooks [1984] forwarding pointers or replication [Nettles and O'Toole 1993]. Barriers enabling incremental collection can be optimized either for throughput or for predictability. Arrays can be represented either contiguously or split into *arraylets* [Siebert 2000; Bacon et al. 2003b; Chen et al. 2003].

For this study, we selected a straightforward defragmenting garbage collection algorithm with standard implementation that is suited to the different GC scheduling strategies that we intend to study in this paper. We implement the collector architecture introduced with Metronome [Bacon et al. 2003b], which is based on the mark-sweep snapshot-at-the beginning Yuasa [1990] collector with weak tri-color invariant. It is mostly noncopying, uses arraylets to limit external fragmentation, and has fully incremental defragmentation using Brooks [1984] forwarding pointers. It uses a segregated free-list allocator with a carefully selected set of supported object size classes to further limit internal fragmentation. We configure Minuteman with barriers optimized for predictability (with minimum branches and activated at all times, even outside of the GC cycle). We build in support for periodic and slack GC scheduling, allowing us to compare the two in otherwise identical configurations. We provide more details below.

3.1. Collection Cycle

Our GC is implemented in a single real-time thread as a loop repeating the following steps. Each iteration is a *GC cycle*.

(1) Wait Until Memory Is Scarce.
(2) Scan Threads.
(3) Mark And Clean Heap.
(4) Clean Threads.
(5) Sweep.
(6) Defragment.

The GC periodically modifies the state of the heap, maintaining particular invariants at each step. The key invariants are formulated in terms of object color (white, grey, or black) and pointer classes (clean, dirty). The meaning of colors is that white is unreachable, and black and grey are reachable. A pointer is dirty if it refers to the old location of an object that moved during defragmentation. A pointer is white, grey, or black if it respectively points to a white, grey, or black object. Brooks forwarding pointers ensure that the mutator always uses the new location of a moved object. Each

Java object has an additional field to store the forwarding pointer. A read barrier makes sure that this indirection is always followed before the pointer is used. For instance, every field assignment `x.f=`*exp* will be translated into `x.forward.f=`*exp*.

3.1.1. Wait Until Memory Is Scarce. At this stage, all existing objects are black, and new objects are allocated black. Any pointer can be dirty, because objects may have been moved during defragmentation in the previous cycle (stage 6). The GC waits until the available memory reaches a predefined threshold. This threshold must be large enough to ensure that there is enough memory left for allocations by the mutator that may occur before the GC can actually free sufficient memory (stage 5). It must also cover the allocation needs of the GC itself during defragmentation (stage 6).

3.1.2. Scan Threads. The graph traversal is started by flipping the values representing black and white and thus making all objects in the heap white. The allocation color is updated to remain black. The GC will reclaim all objects that were already unreachable when the traversal started. Tricolor invariants are enforced by a combination of the Yuasa [1990] "deletion" barrier and the Dijkstra "insertion" barrier [Dijkstra et al. 1978]. The Yuasa barrier enforces the weak invariant that every white object pointed to by a black object is also reachable from some gray object through a chain of white objects. Before every store to a pointer location in the heap, the Yuasa barrier captures the old pointer being deleted from that location and marks the target of the pointer gray. This prevents breaking a chain of references from a gray object to a white object.

To avoid long GC pauses, scanning of thread stacks for pointers is interruptible. This means that some thread stacks are logically gray (still need to be scanned) while others are logically black (have already been scanned). Before tracing begins all stacks are gray and contain only white references. Scanning each stack flips it from gray to black. Note that under the weak tricolor invariant even black thread stacks can subsequently acquire white references by loading them from the heap. The weak tricolor invariant means that such references remain reachable via a chain of white pointers from some gray object. However, a pointer from a gray stack to a reachable (i.e., referenced from black) but still white object may be the only pointer preventing that object from being collected. Deleting such a pointer from the gray thread stack would violate the weak invariant. This could be prevented by applying a Yuasa barrier to mutation of thread stacks, but that would be prohibitively expensive since stack (and register) mutations are very frequent. Instead, using a Dijkstra barrier prevents storing of white references into the heap so that no reference from black objects to white objects can be created. This barrier marks gray the target of any pointer stored to the heap. It applies only to heap stores, which are much less frequent than stack/register mutation. The Dijkstra barrier makes scanning of stacks for pointers interruptible between threads but scanning a single thread must be atomic. Stack scanning is fast in Ovm due to the use of pointer stacks [Baker et al. 2009]. Threads can be scanned independently as Java ensures that communication between threads is done through shared heap locations (the same would not be true in C where pointers to the stack can be created).

3.1.3. Mark And Clean Heap. Gray objects are scanned one by one, marking all reachable white objects gray, and marking all scanned objects black. Scanning objects, including arrays, is fully incremental. As objects are scanned, all dirty pointers they contain are fixed to point to the forwarded locations, so that after the whole heap is scanned, all pointers in black objects are clean. When marking, the new location of an object is marked. The old location, if any, is left white. To prevent dirty pointers from spreading into already-scanned objects, either from unscanned objects or from the stacks, dirty pointers are also fixed in the write barrier. This is strictly necessary only until the stacks are clean (see next step), but the code to do so is permanently compiled into the

write barrier for predictability. In any case, the mutator must be prepared to see both clean and dirty pointers for the same object, and thus even pointer comparison must follow the Brooks forwarding pointers. Global variables are scanned similarly to gray objects; global data is always assumed live. Once the heap is cleaned, there are only white and black objects. White objects are garbage and black objects are assumed live, although some of them might have died since the GC cycle started.

3.1.4. Clean Threads. Because the heap is now clean, no more dirty pointers can be loaded to the thread stacks, and thus the stacks can be fixed to point to the forwarded locations of moved objects. Fixing is again atomic with respect to each thread, but can be interrupted after each stack is fixed.

3.1.5. Sweep. Now the mutator only has access to black objects and to new locations of objects. The white objects (garbage and old locations of objects evacuated during the last defragmentation) are reclaimed and the memory they occupy is made ready for reuse. This involves some housekeeping of free memory and, since Java requires all allocated storage to be zeroed, also zeroing the freed memory. While some collectors zero memory on allocation on behalf of the mutator, we do so at sweep time by the collector. Thus, high-priority tasks are not delayed by zeroing due to earlier allocations by tasks with a lower priority. Also, zeroing at sweep time makes it easier to attribute the resulting overhead to the collector. The sweep operation can be interrupted at almost any time. This is achieved by making sure the allocator cannot see memory that is not yet ready for reuse. Memory organization is relatively complex in order to reduce fragmentation and minimize the amount of work done during defragmentation. All memory is divided into equal-sized pages (2 kilobytes). These pages can be used directly as arraylets or for large nonarray objects, or they can be further partitioned for allocation of small objects. Large nonarray objects can create both external and internal fragmentation, but they are rare in real applications.[1] With arraylets, arrays cause no additional fragmentation over small objects: all space in the arraylets is used by array data, the remaining array data (smaller than an arraylet) is stored in a small object called the *spine*, together with pointers to the arraylets. Small objects are rounded up to a predefined set of object sizes. Within a single page, only small objects of the same (rounded-up) size can be allocated. Page size is not always a multiple of (rounded-up) object size, and thus there is some wasted space on each page. The amount of this per page internal fragmentation versus per-object internal fragmentation can be controlled by tuning the number of supported object sizes. Moreover, this amount is proportional to the number of live objects and is relatively small. Yet another kind of fragmentation is caused when only a few objects in a page die, while the others are still live. These empty slots can be reused, but not for another object size. Thus, an unlucky sequence of allocations can lead to running out of memory in a system with little memory actually used. This is why the GC implements defragmentation. Free slots are organized in general as segregated free lists. However, when the allocator needs a new page for small object allocation, it first allocates from that page sequentially by incrementing a pointer (so-called *bump-pointer* allocation). Thus, free lists are only initialized during sweep by the GC, as it discovers pages that contain both live and dead objects.

3.1.6. Defragment. The heap now contains only black objects, the allocation color is still black, and all (live) pointers in the system are clean. Each object has only a single copy. However, pages containing small objects may be fragmented. There may be a lot of unused space in pages reserved for small objects of a particular size (size class of pages).

[1]We measured that the largest nonarray object uses only 432 bytes in the DaCapo, SPECjvm98, and pseudo-JBB application benchmarks.

This space can be reused only for objects of that size, but a request to allocate an object of a different size may lead to running out of memory. Defragmentation is only started if the amount of free memory is below a defined threshold. Starting defragmentation too early is a waste of time, and starting defragmentation too late can be a problem as it temporarily reduces the amount of free memory. Defragmentation starts with pages from a size class for which most memory can be freed by defragmenting: moving objects from less occupied pages to more occupied pages (this direction of copy is to minimize the number of copied objects for a given number of freed pages). Each size class has a list of nonfull pages, which is also used by the allocator. The defragmenter incrementally sorts this list in order of decreasing occupancy. Then it copies the objects from the tail of the sorted list (least occupied pages) to the head (more occupied pages). This operation is incremental (except for the copying of each small object) and does not harm the mutator. Termination is somewhat subtle, since the mutator may quickly re-use the space intended as target for evacuation. In that case, the defragmenter bails out and moves to the next size class. In the worst case, the defragmenter would bail out from every size class (by default, the GC has 28 size classes). After copying an object, the defragmenter (still atomically) updates the forwarding pointers, so that the new copy of the object points to itself and the old copy points to the new copy. The evacuated pages will become reusable for allocation during the sweep phase of the following GC cycle.

4. SCHEDULABILITY ANALYSIS

The role of schedulability analysis is to answer the question of whether all tasks in a given set can meet their deadlines. To obtain a precise answer, all costs and sources of interference between tasks must be precisely accounted for. In the literature, it is traditional to abstract most of these costs by a small number of input parameters that are assumed to be obtained externally (either through analysis or careful measurements) and focus on the essence of the scheduling problem. We follow suit and start from well-known schedulability tests which model periodic tasks with fixed priorities and without blocking [Joseph and Pandya 1986; Fidge 1998]. Additional parameters such as interference due to mutual exclusion can be added but are mostly orthogonal. We have chosen to study scheduling of garbage collection in the context of fixed-priority preemptive scheduling strategy as this is the approach supported by commercial implementations of the RTSJ. Accommodating dynamic scheduling algorithms, such as earliest deadline first, would require changes to the response time analysis of mutator threads and a tighter integration with the underlying scheduler.

In the model, tasks are identified by integers, which are also their priorities, 1 (highest) to n. We assume that the worst-case execution time of each instance of task i is its cost C_i, a value that may be obtained either by static or dynamic analysis [Wilhelm et al. 2008]. Each task has a deadline T_i which reflects the application's timing constraints. For simplicity we assume that the task's period and deadline are identical. The response time of task, R_i, is the longest duration between the time a task becomes runnable and the time it completes executing. Thus for a schedulable task set $C_i \leq R_i \leq T_i$. The response time of a task i can be computed by assuming that all tasks of higher priority than i are released for execution simultaneously with task i and is computed by Equation (1):

$$R_i = C_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j \right) \tag{1}$$

Table I. Input Parameters for Schedulability Analysis

C_i	[seconds]	computation time	task
T_i	[seconds]	period	task
A_i	[bytes]	allocation	task
G_i	[seconds]	GC work generated	task
H	[bytes]	heap size	system
$L_{\max}$	[bytes]	live memory	system
T_{gc}	[seconds]	GC cycle duration (period)	system
G_0	[seconds]	GC cycle overhead	system

Thus the response time R_i is the cost of the task i and the sum of the preemption costs of higher priority tasks. Note that the term $\lceil \frac{R_i}{T_j} \rceil$ is the maximum number of times task j could be released during a single computation of task i. As all tasks $j, 1 \leq j \leq i-1$ have a higher priority than i, they will preempt i at each of their release, hence prolonging R_i by C_j at each release. The recurrence is solved iteratively as follows:

$$R_i^0 := 0$$
$$R_i^{n+1} := C_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \right).$$

The system is schedulable if and only if for every task i the equation converges to a finite fixed point R_i, such that $R_i < T_i$.

4.1. Schedulability and Garbage Collection

Incorporating the garbage collector task into schedulability analysis requires enriching the execution model because the cost and period of the collector is dependent on the behavior of other tasks. So, in addition to the costs C_i and deadlines T_i of individual tasks, information about the amount of work created by the application and the characteristics of the GC must be provided. Under a fixed priority scheduling regime, the priority of the garbage collection task is a key design decision. In slack-based GC, the collector runs at the lowest priority. It can thus be preempted by all other tasks. On the other hand, a periodic GC runs at the highest priority, preempting all other tasks in the system. To bound the periodic GC's impact on other tasks, it voluntarily yields the processor at regular intervals. For both slack-based and periodic GC, the GC task is triggered periodically. We assume that the period of the GC task, T_{gc}, is provided by the user. It must be sufficiently large to allow for a complete GC cycle. The work to be done during any GC cycle depends on the memory operations performed by mutator tasks, such as allocations, and loads/stores to reference variables. The constant G_i captures the worst-case amount of GC work that a single instance of task i can generate. The constant G_0 is the upper bound on the per-cycle GC work that is not dependent on the operations performed by the application. This covers operations such as scanning of stacks and global variables. The maximum heap size, H, is a parameter chosen by the user. $L_{\max}$ is the maximum amount of live memory at execution. It can be obtained either by program analysis or through measurement. The upper bound on allocation per invocation of a task is A_i. The input parameters are summarized in Table I.

Schedulability of a set of tasks with GC requires that the application tasks and the GC meet the following three conditions.

T1. mutator tasks meet their deadlines.
T2. GC meets its deadline and keeps up with tasks that use memory.
T3. the system does not run out of memory.

The conditions can be formalized as three tests that can be be checked based on the above mentioned input parameters. The formulation of tests **T1** and **T2** is dependent on how the GC is scheduled, but **T3** can be formulated independently of GC scheduling. We can find an upper bound on the GC cycle duration that ensures that all allocation requests can be fulfilled. We use a bound based on that of Robertz and Henriksson [2003]:

$$A_{\max} \leq \frac{H - L_{\max}}{2}. \tag{2}$$

In Equation (2), $A_{\max}$ stands for the the maximum amount of allocation performed by the application tasks during a GC cycle. It is easy to see that the condition is necessary for the system not to run out of memory as objects that become unreachable during the GC cycle in which they are allocated (they are called floating garbage) can only be freed by the end of the subsequent GC cycle. Previous work has shown that this is a sufficient condition [Robertz and Henriksson 2003; Schoeberl 2010]. The intuition behind the proofs is as follows. The worst case occurs when the amount of live memory is $L_{\max}$ for all cycles. The maximum amount of floating garbage is the maximum allocatable memory in the previous cycle ($A_{\max}$) and the maximum permissible amount of memory ($A_{\max}$) is allocated in the present cycle. The system does not run out of memory as long as $A_{\max} + A_{\max} + L_{\max} \leq H$, and thus Equation (2) is sufficient.

The values of $A_{\max}$ and the related constant $G_{\max}$, which stands for the maximum GC work, can be derived from the input parameters. When T_{gc} is a multiple of the hyperperiod ($\text{lcm}_{i=1..n}(T_i)$) and all tasks are started simultaneously, a simplified equation can be used:

$$G_{\max} = G_0 + \sum_{i=1}^{n} \frac{T_{\text{gc}}}{T_i} G_i$$

$$A_{\max} = \sum_{i=1}^{n} \frac{T_{\text{gc}}}{T_i} A_i.$$

Aligning T_{gc} to the hyperperiod may lengthen GC cycles and thus lead to higher memory requirements. The following equations consider the general case. Making no assumptions about how the generation of GC work is distributed within each task, any task that runs, even partly, during a GC cycle, will contribute its entire GC work to that cycle:

$$G_{\max} = G_0 + \sum_{i=1}^{n} \left(\left\lceil \frac{T_{\text{gc}}}{T_i} \right\rceil + 1 \right) G_i \tag{3}$$

$$A_{\max} = \sum_{i=1}^{n} \left(\left\lceil \frac{T_{\text{gc}}}{T_i} \right\rceil + 1 \right) A_i. \tag{4}$$

The benefit of these equations is that T_{gc} can be smaller than the hyper-period, which reduces the memory requirement H. These equations do not require tasks to be started simultaneously.

4.1.1. Slack GC. Under slack-based scheduling, the GC does not interfere with the other tasks in the system. This simplifies test **T1** because mutator task deadlines can be checked as if there was no GC. More precisely, since the GC is the lowest priority task, it will not influence their response time and Equation (1) can be used as is. Computing the response time of the GC tasks, R_{gc}, is required to check that test **T2**

holds. Equation (5) simply states that R_{gc} depends on the maximum amount of GC work per cycle and the time required by all other tasks.

$$R_{gc} = G_{max} + \sum_{i=1}^{n} \left(\left\lceil \frac{R_{gc}}{T_i} \right\rceil \cdot C_i \right). \tag{5}$$

This recurrence is similar to the tests for mutator periodic tasks. The iterative process to find R_{gc} is also similar, except that we can start with $R_{gc}^0 := \sum_{i=1}^{n} C_i$.

4.1.2. Periodic GC. Under periodic scheduling, the GC runs as the highest priority task in the system, but rather than doing all of its work at once it cooperatively yields the CPU to the mutator task. The GC cycle is divided into fixed-size time quanta. Each quantum may be allocated either to the mutator tasks or to the collector. The allocation is done statically, independently of the underlying real-time scheduler, and gives rise to what we refer to as an *MC schedule* as it is expressed by a a sequence such as `CMCMCMCMM` where `M` represents a mutator quantum and `C` a collector quantum. The choice of quantum size and MC schedule are key parameters for schedulability analysis. A typical implementation will block the collector thread at the start of a mutator quantum, and unblock it at the end of the quantum. When there is no GC work to be done, the mutator may be allowed to use a collector quantum.

To determine the response time of mutator tasks, it necessary to account for the GC interruptions, and conversely the response time of the GC requires accounting for the time yielded to the mutator. We do this with the notion of minimum utilization devised by Cheng [2001]. The function $mmu(t)$ gives the minimum mutator utilization (MMU) for any window of length t; that is to say, what percentage of the CPU time is given to mutator tasks in any interval of t seconds. The function $mcu(t)$ is the minimum collector utilization for a window t. Both functions depend on the MC schedule and can be computed by a brute force algorithm. Consider the MC schedule `CMM` with a quantum size of $100\mu s$, the $mmu(t)$ for a window $t = 200\mu s$ is $\frac{200-100}{200} = 50\%$ because the worst case is that the collector interrupts the mutator for one quantum. But $mcu(200) = 0$ because the worst case for the collector occurs when the mutator runs for two consecutive quanta. As t grows, the functions will converge to $mmu(t) = 1 - mcu(t) = u$, where u (*target utilization*) is the ratio of mutator quanta `M`s in a cycle. If q is the length of a time quantum, then $mcu(t)$ and $mmu(t)$ are linear functions within each interval $[nq, (n+1)q]$ for integer n. The task response time depends on mmu and on the maximum amount of work per cycle for **T1**:

$$R_i = C_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \right) + \min \left\{ (1 - mmu(R_i)) \cdot R_i, \left\lceil \frac{R_i}{T_{gc}} \right\rceil \cdot G_{max} \right\}. \tag{6}$$

The time added by GC (the last term) is the maximum time the GC can take during R_i, which is $(1 - mmu(R_i))R_i$. However, as expressed by the second argument of min, if the GC cycle is shorter than R_i and the GC does all its work in a cycle, it will no longer be taking time from the mutator.

The worst-case response time for the GC, needed for test **T2**, can be calculated as follows.

$$\text{Let } R_{gc} \text{ be the smallest } t \text{ such that } t \cdot mcu(t) \geq G_{max} \text{ and } t \leq T_{gc}. \tag{7}$$

The recurrence cannot directly be solved iteratively, because mcu is not a monotone function over its full range.

The Metronome collector [Bacon et al. 2003b] and its commercial implementations invented the following approach for specifying the MC schedule. Rather than defining

a schedule for the entire GC cycle, they define MC patterns, which are then repeated to cover the entire running time of the application. The length of the pattern and the quantum defines a window w. Since our implementation mimics Metronome, we can leverage this to compute R_{gc}. Because the MC schedule is constructed by repeating the MC pattern, $mcu(t)$ is constant for any t being a multiple of the window size w. The constant is equal to the ratio of collector quanta within the MC pattern. To calculate R_{gc}, we thus first find a solution t_w in multiples of the window size w:

$$t_w = w \left\lceil \frac{1}{w} \frac{G_{\max}}{1-u} \right\rceil. \tag{8}$$

We now know that the smallest t such that $t \cdot mcu(t) \geq G_{\max}$ is in the interval $(t_w - w, t_w]$. Given that $mcu(t)$ is linear within time interval $(t_w - w, t_w]$ and that $mcu(t_w) = mcu(t_w - w)$, it follows that the solution t is a multiple of the quantum size. There is only a limited small number of quanta per window, so we can easily enumerate $t \cdot mcu(t)$ for all such t and choose the best one.

4.1.3. Hybrid GC. Hybrid scheduling (originally devised to enable multiprocessor systems to offload collector work onto hardware threads [Auerbach et al. 2008]) is an attempt to combine the periodic and slack-based scheduling approaches by allowing the collector task to take additional quanta when all mutator tasks are idle. Response time of mutator tasks is computed by Equation (6), as with periodic GC scheduling. The intuition is that when the GC takes a mutator quantum, its priority is decreased to be lower than that of mutator tasks and thus if the mutator needs the CPU it can simply preempt the collector. The advantage of hybrid GC scheduling is that when there is slack, T_{gc} and $G_{\max}$ can be smaller than with periodic, because the GC work can finish earlier for the same heap size. The GC response time for test **T2** is defined as follows.

$$\text{Let } R_{\text{gc}} \text{ be the smallest } t \text{ such that } t \cdot mcu(t) + slp(t) \geq G_{\max} \text{ and } t \leq T_{\text{gc}}, \tag{9}$$

where $slp(t)$ is the guaranteed amount of slack in any window of length t provided that a hybrid collector takes all quanta it can during this time, which is in turn reflected by the term $t \cdot mcu(t)$. The function $slp(t)$ is as follows.

$$\text{Let } slp(t) \text{ be the largest } g \text{ such that } \exists r \leq t \quad \text{such that} \tag{10}$$

$$r = g + \sum_{i=1}^{n} \left(\left\lceil \frac{r}{T_i} \right\rceil \cdot C_i \right) + \min \left\{ (1 - mmu(r)) \cdot r, \left\lceil \frac{r}{T_{\text{gc}}} \right\rceil \cdot G_{\max} \right\}.$$

Intuitively, we are looking for the maximum amount g of GC work we can do within time t considering that the GC can be preempted by any task. The recurrence equation for r comes directly from Equation (6) and is solved in the same fashion; r is the response time of an imaginary lowest-priority task with computational cost g. The GC response time can be found by exhaustive search in integer numbers or approximately, using Equation (9), by checking for several t, $G_{\max} \leq t \leq T_{\text{gc}}$, if $t \cdot mcu(t) + slp(t) \geq G_{\max}$, and then taking a minimum of such ts. Any upper bound found is safe, but the selection and number of tested ts affects how close the bound is. Calculating $slp(t)$ also requires an exhaustive search or approximation. We use a binary search, iteratively solving Equation (10) at each step.

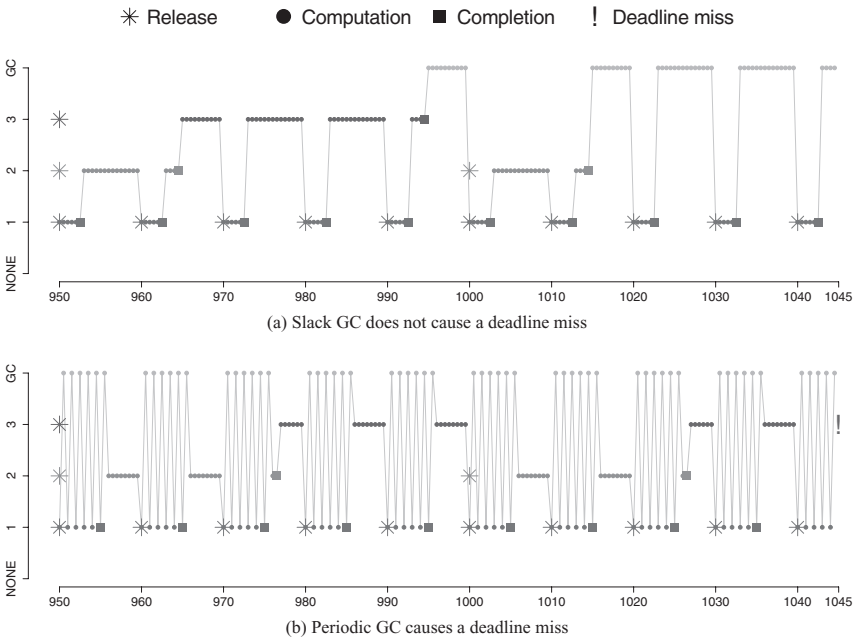
4.2. Schedulability Case Studies

While the set of applications that can be scheduled by periodic and slack-based GC scheduling policies overlap, neither is strictly superior to the other. We prove this by exhibiting two examples: in one the system is schedulable with a periodic GC and

i	T_i	C_i	A_i	G_i
1	10	3	72	1
2	50	9	302	5
3	95	21	256	4

L_{max}	G_0	H	T_{gc}
300	10	25500	730

Quantum size: 0.5
Window size: 10
MC pattern: MCMCMCMCMCMCMMMMMMMM



(a) Slack GC does not cause a deadline miss

(b) Periodic GC causes a deadline miss

Fig. 2. *Slack.* Task 3 with periodic GC misses a deadline at 1045ms.

not with a slack-based GC, while in the other it is schedulable with a slack-based GC but not a periodic GC. Hybrid scheduling is strictly more powerful than periodic, but as shown by one example does not subsume slack-based. On the other hand, slack-based scheduling does not subsume hybrid, either: we show an example of a system schedulable with hybrid scheduling, but neither schedulable with periodic nor slack scheduling.

For graphical demonstration of the different types of scheduling we have implemented a simple GC scheduling simulator. In the simulator all mutator threads in all of their releases take the worst-case execution time, allocate the maximum amount of memory and generate the maximum amount of GC work. The initial task offsets are configurable (and zero by default). If there is no accumulated GC work when the collector starts running during a collection cycle, such as during the first cycle of the simulation, the collector does not do anything in that cycle. If there is any accumulated work, it performs that amount of work plus G_0. The reclaimed memory is made available when the collection work in a cycle is finished. The allocation within a mutator release is bunched at the beginning and the generation of GC work is bunched at the end (indeed, these choices are arbitrary). In the following we show examples where non-schedulability detected by the formula is also triggered in the simulator. The simulator works at the resolution of the time quanta of the periodic GC; its implementation is a straightforward application of the GC scheduling algorithms.

4.2.1. Slack. The first case study is Figure 2, adapted from Henriksson [1998], which has three mutator tasks labeled 1 to 3 ordered by priority (with task 1 having the

highest priority). We start by testing if the mutator tasks always meet their deadlines (**T1**). The iterative solution of Equation (1) based on the periods T_i and costs C_i gives the response times, R_i and confirms that the tasks are schedulable ($R_i < T_i$ in all cases):

$$\begin{aligned} R_1 = 3 &< 10 = T_1 \\ R_2 = 15 &< 50 = T_2 \\ R_3 = 45 &< 95 = T_3. \end{aligned}$$

Next, we test if GC can keep up with the mutator tasks. We compute the bound on the GC work $G_{\max}$, which, in turn, requires the GC cycle overhead G_0, G_i, and the period of the GC task, T_{gc}. From Equation (3) we get

$$G_{\max} = 10 + 74 \cdot 1 + 16 \cdot 5 + 9 \cdot 4 = 200.$$

Solving Equation (5) (**T2**) we get $R_{\text{gc}} = 719$. As this is less than the GC's period of 730, the GC can keep up with mutator. Last, we must check that the system does not run out of memory (**T3**). For this we need to know the available heap size H, the allocation rate of the tasks, A_i, and the upper bound on live memory $L_{\max}$. From Equation (3) we get $Amax = 12464$, which satisfies Equation (2):

$$12464 < \frac{25500 - 300}{2} = 12600$$

The system thus does not run out of memory. With periodic GC, test **T1** fails because task 3 is not able to complete its work before its deadline, as computed by Equation (6). The same test fails for the hybrid approach. Figures 2(a) and 2(b) show simulations of this workload with a slack-scheduled and periodic GC. Figure 2(a) clearly shows how the slack-based GC's work is bunched towards the end of the period after tasks 2 and 3 are done with their work. It also shows the slack-based GC being preempted by task 1 and 2. In this example the CPU is fully utilized. Figure 2(b) shows that periodic GC is active most of the time (according to its MC schedule). The deadline miss occurs quite late in the run because there is no garbage at the start of the simulation and thus the GC only kicks in once some work has been generated. The deadline miss occurs at time 1045. The periodic GC runs frequently, interrupting the mutator's progress, and decreasing the response time of the mutator tasks: tasks 1 and 3 get less work done, and task 3 misses its deadline.

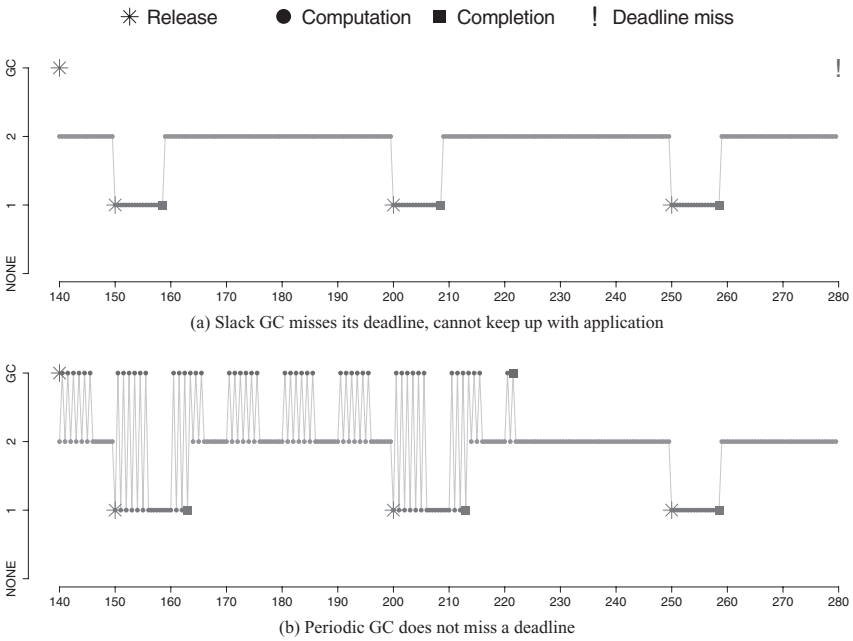
4.2.2. Periodic. Now, consider our second case study described in Figure 3. We start with a periodic GC, test **T1**, so Equation (6) gives task response times of 14 and 927. The mutator tasks are thus schedulable. The upper bound on T_{gc} is 150 given the available heap size. The system thus will not run out memory during the GC cycle. For **T2** we need to solve Equation (7): $R_{\text{gc}} = 128$, which is well within the GC cycle length. This configuration is not schedulable by a slack-based GC because the GC cycle $T_{\text{gc}} = 140$ is smaller than the cost of task 2, $C_2 = 490$. This means that even if there is enough slack, that slack comes too late—the system will have run out of memory before the slack-based GC could help. Figure 3(a) shows the critical part of the schedule. The GC misses its deadline at time 280 because task 2 is still running. Figure 3(b) shows that in this case the periodic GC's interruptions do not cause the mutator to miss their deadline, and because the GC is able to preempt the long running task 2 it collects the garbage in time.

4.2.3. Hybrid. Finally, consider the example of Figure 4 with a hybrid GC. The system is schedulable with mutator task response times (**T1**, Equation (6)) of $R_1 = 6$, $R_2 = 18$, $R_3 = 324$ and GC response time (**T2**, Equation (9)) $R_{\text{gc}} = 469$. The maximum GC

i	T_i	C_i	A_i	G_i
1	50	9	302	5
2	980	490	65	4

L_{max}	G_0	H	T_{gc}
300	10	3000	140

Quantum size: 0.5
Window size: 10
MC pattern: MCMCMCMCMCMCMMMMMMMM



(a) Slack GC misses its deadline, cannot keep up with application

(b) Periodic GC does not miss a deadline

Fig. 3. *Periodic.* The slack GC misses a deadline at 280ms (does not run at all in this cycle).

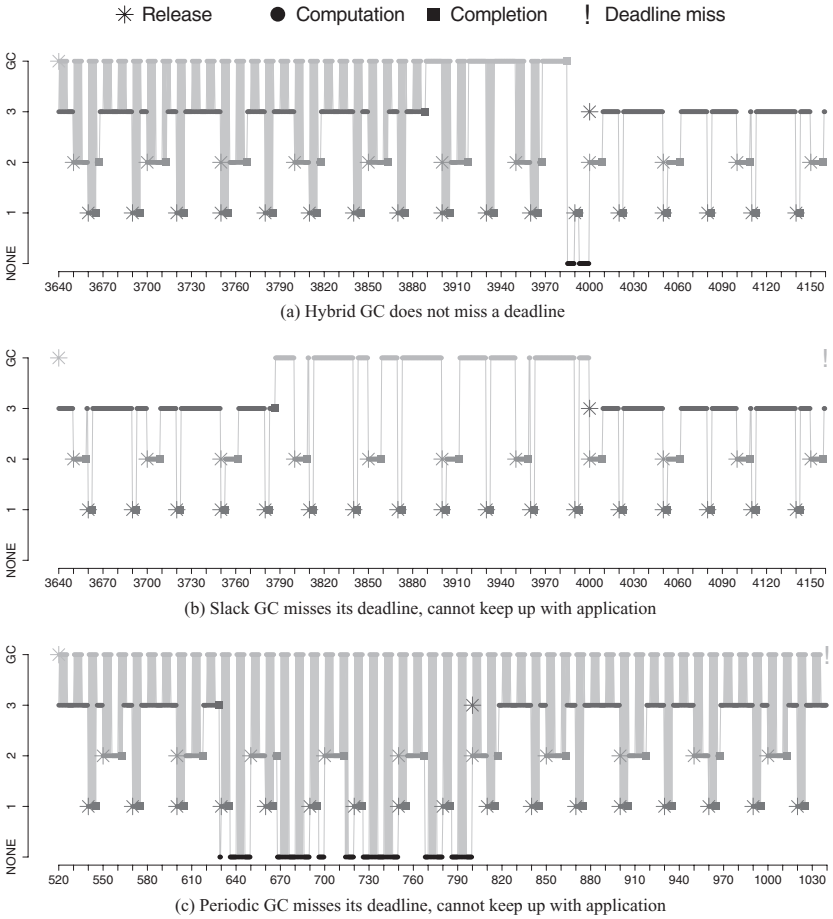
cycle length that ensures not running out of memory is (**T3**) 550. The example is not schedulable with slack GC or periodic GC, as with both the GCs miss their deadlines. The periodic GC misses a deadline because it cannot use slack quanta that belong to the mutator: there is slack time wasted, despite GC work left to do. The slack GC misses a deadline because of very uneven distribution of slack in the schedule. Note that computation time of task 3 is as much as 130. Its response time with a slack-based GC is 187. There can be two releases of task 3 within one GC cycle, thus there can be only 156 of slack time within a cycle, but the maximum GC work generated per cycle is 177. Over longer time intervals, the amount of slack would be sufficient: there is enough slack time in a time interval of two adjacent collections to finish two collections, and even more so for longer intervals. The hybrid GC can re-organize this slack time by stealing mutator quanta, so that it meets the GC deadline. The slack GC cannot, and thus fails. Note that increasing the heap size, and thus allowing a longer GC cycle, could make the system schedulable also with the slack GC.

Figure 4(c) shows the critical part of the schedule where the periodic collector fails (at the given time scale, the interruptions by the periodic collector appear very close). The periodic GC fails to use slack time in the middle of the schedule, because the slack is allocated to the mutator. The hybrid scheduler uses the slack time, and thus meets the deadline (Figure 4(a)). Figure 4(b) shows the critical part of the schedule where the slack scheduler fails in the simulator (it fails in a different part of the schedule). The slack collector in the part of the schedule shown has more work to do than the hybrid

i	T_i	C_i	A_i	G_i
1	30	3	72	5
2	50	9	302	5
3	400	130	256	4

L_{max}	G_0	H	T_{gc}
300	10	12000	520

Quantum size: 0.5
Window size: 10
MC pattern: MCMCMCMCMCMCMMMMMMMM



(a) Hybrid GC does not miss a deadline

(b) Slack GC misses its deadline, cannot keep up with application

(c) Periodic GC misses its deadline, cannot keep up with application

Fig. 4. *Hybrid.* Periodic and slack GCs miss their deadlines.

collector, because it starts later. Hence it misses its deadline, although it does even more GC work in this interval than the hybrid scheduler.

5. REAL-TIME COLLISION DETECTOR

For the empirical evaluation, we use the Real-Time Collision Detector (RCD) benchmark [Kalibera 2009b]. The key component of this benchmark is a periodic real-time task that detects potential aircraft collisions based on radar frames. Each invocation of the task takes a new radar frame which contains up-to-date locations of aircraft with their identifications, computes current motion vectors based on the previous known locations of the aircraft, and then uses the motion vectors to detect collisions. The detection algorithm has two stages: in the first stage the detector identifies smaller two-dimensional quadrants containing multiple aircraft, while in the second stage it performs three-dimensional detection in these quadrants. We use a version of the

benchmark that pre-simulates all radar frames before the detector is started, to minimize interference and unwanted time dependencies. We use the RTSJ scheduling and timer API, but perform all allocation in the heap (no scopes or immortal memory). For the purpose of our evaluation, we modified the original benchmark to support multiple parallel detector tasks. Each parallel task uses the same detector algorithm, but different tasks run at different priorities, have different periods, and accordingly process different numbers of radar frames. The benchmark reports response times of the collision detector tasks. The response times are measured from idealized release times (no jitter, ideal timer) to the actual completion time within a task release. The idealized release times are defined by absolute start time, which is rounded to avoid phase-shift of real timers [Burns et al. 1995], and by the period. We use two parallel detector tasks for this evaluation.

We use two different workloads, S and P, which are designed to favor, in turn, slack and periodic/hybrid scheduling, and to exercise different parts of the memory allocator, garbage collector, and collision detector. The S workload has 19 aircraft with no collisions and detector tasks with periods of 8ms and 16ms. The P workload has 40 aircraft with collisions and detector tasks at periods of 10ms and 2s. To make the workload more challenging to the garbage collector, we add a simple loop that allocates a variable number of objects to the longer running task. In the P workload, we use a simpler configuration of the allocator than in S, but also a smaller heap. In the P workload, we allocate a million 64-byte objects per task invocation. Pointers to the objects are stored into a cyclic buffer of 0.5 million objects, so that except for the first invocation, 0.5 million old objects and 0.5 million new objects die after each invocation. In the S workload, instead of a constant object size, the allocations follow a predefined pattern, parameterized by maximum allocation request size and an allocation size increment. The number of allocated objects per invocation is 1,000 and the cyclic buffer for pointers has 10,000 elements. Thus, after 10 invocations 1,000 of these objects die at each invocation. In both workloads, we run 10,000 invocations of the task with the longer period. To focus on steady state performance, we skip the first 2,000 and the last 100 invocations. The initial invocations are subjected to initialization noise, while the final invocation suffers noise due to shutting down individual detector tasks. For the task with 2s period, we only skip 10 initial and 10 last invocations.

The periodic and hybrid schedulers use a 10ms window, 500μs quanta, a 1ms maximum pause time, and $u = 0.7$ target minimum mutator utilization (or MMU). These are the defaults that are also used by IBM WebSphere. The MC scheduling pattern is `MCMCMCMCMCMCMMMMMMMM`. The slack scheduler is configured so that GC runs at a lower priority than the real-time detector tasks. The same priority is used as the lower priority of the hybrid scheduler. Our experimental platform is Ubuntu Linux 2.6.31, which already has real-time support by default, running on an Intel Core2 CPU at 2.83GHz with 6M Level 2 Cache. We built Ovm with the GNU GCC 4.4 compiler.

6. EMPIRICAL EVALUATION

The schedulability analysis of Section 4 is performed in an abstract model of computation to keep the formal treatment focused and compact. One could add features such as lock-based concurrency control and priority inversion avoidance protocols which are supported in Java [Bollella et al. 2000] and have well-understood formal treatment. Adding dynamic thread creation would require admission control but would not change our analysis. Getting values for the input parameters to the scheduling tests is more tricky, especially when running on modern architectures that add some unpredictability with features such as caches and pipelines. Estimating the cost C_i and G_i requires a precise model of the architecture and of the flow of control through the program. The state-of-the-art in worst-case execution time analysis [Wilhelm et al. 2008] suggests

that it may be possible to obtain those values, but our practical experience with commercial tools suggests that analyzing the rather complex code generated by a virtual machine will require further research. An alternative would be to use a simpler processor where all operations have fixed, fully predictable costs [Pitter and Schoeberl 2010]. Allocation costs can be bounded by static analysis [Garbervetsky et al. 2009] but there are no tools for computing precise bounds on the amount of GC work created by a task.

This section will side-step these issues and provide the first empirical evaluation of the different GC scheduling approaches on an efficient virtual machine and with realistic workloads. One of the goals of evaluating actual implementation is to reveal any unforeseen overheads (such as excessive barrier costs, pathological cache behavior, and operating system costs). Furthermore, the study gives a feeling for the average performance of the different approaches which is important for soft real-time applications. We also demonstrate experimentally that there are cases where none of the scheduling strategies is strictly better than the others.

We start with Section 6.1, where we illustrate the complexity of evaluating real-time applications with the minimum mutator utilization metric. Section 6.2 measures observed response times, which we argue are a better metric for comparing GCs. Section 6.3 shows the impact of adding non-real-time activity to a real-time system. Finally, Section 6.4 and Section 6.5 detail the overheads and relative costs of the GC algorithms.

6.1. Minimum Mutator Utilization

The minimum mutator utilization, or MMU, is a metric originally proposed to measure the quality of work-based real-time garbage collectors [Cheng 2001; Cheng and Blelloch 2001]. The observation was that maximum pause times are not meaningful because a collector with very short but very frequent pauses can be worse from the point of view of a real-time mutator task than a collector with longer, but less frequent, pauses. The idea of the MMU metric is thus to measure, for a time window of size t relevant to the mutator tasks, the worst-case mutator utilization. This gives a lower bound on the number of CPU cycles that the mutator can use in any given interval of size t.

While MMU works well for its original intent, the temptation to apply it more widely to measure different kinds of real-time garbage collectors should be resisted.

We illustrate the discussion with Figure 5, which shows the MMU of the RCD benchmark, P workload with a 10ms period. The MMU is computed for window sizes ranging from 100μs to 1s. For window sizes smaller than the longest GC pause, the MMU will be zero and, as the window size increases the value of the MMU will eventually converge towards the target MMU (which is a parameter of the Metronome algorithm). Higher MMU values are better as they indicate a larger portion of the CPU cycles are available to the mutator tasks. One intended use of Figure 5 is for developers to look the MMU value for a window size matching the period of a mutator task and check if there is enough CPU for the task to complete its work.

The problem is that MMU is unaware of scheduling. In particular, the metric is computed without regard to slack which is particularly awkward when looking at algorithms that exploit slack in the mutator threads. A naïve reading of Figure 5 would suggest that both slack and hybrid scheduling are much worse than periodic GC scheduling. Hybrid and slack practically overlap and, at best, offer a guarantee of 10% utilization to mutator tasks. Comparing this to the 70% utilization offered by a periodic GC, there is no question which GC scheduling approach users should choose. This conclusion is certainly wrong in the general case as we have shown that periodic scheduling can fail to schedule workloads that are schedulable with a slack-based scheduler. Even in the particular example of Figure 5, the MMU is misleading. It does not account for the fact that slack-based GC (and hybrid) runs when the mutator tasks are not runnable. From the point of view of the mutator, slack-based scheduling ensures

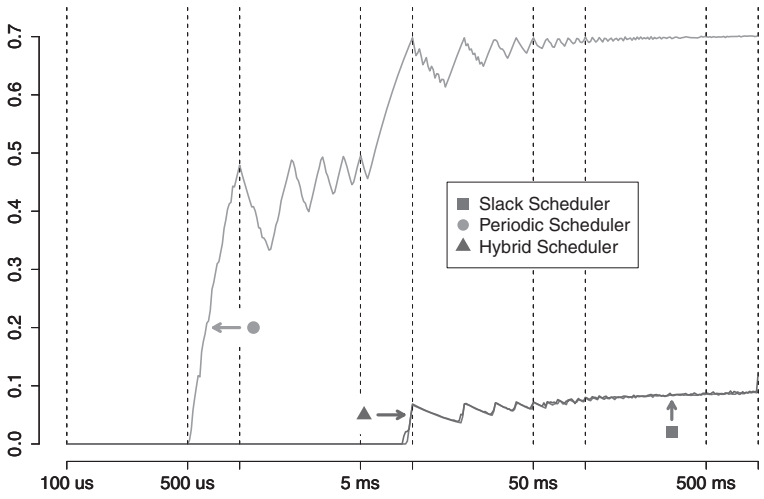


Fig. 5. *MMU.* Computing minimum mutator utilization of the RCD benchmark. The x-axis indicates the window size (ms, log scale). The y-axis is the percentage of CPU cycles available to the mutator (MMU).

100% utilization as the mutator can run whenever it needs to without interruption. Another way to interpret Figure 5 is to observe that periodic GC converges to 70% utilization which is the requested value. But in this benchmark the mutator has about 1ms of computation every 10ms period. Thus the MMU for slack-based is a measure of the available slack which the GC will steal and thus finish its cycle earlier.

An additional drawback of MMU is that an implementation may not account for all GC overheads. Allocation costs and the barriers inserted by the compiler around memory operations are charged to the mutator threads. However, these costs are a major feature of GC design with major impact on performance. This is not an issue with our experiments as Minuteman allows us to retain the same mutator overheads and only vary the scheduling policy, but for instance a work-based garbage collector would technically have MMU of 100% as all of the GC work happens in the mutator thread.

The MMU is thus not the metric we will choose for comparing different GC scheduling strategies. Application specific metrics can be more appropriate [Printezis 2006], but for our purposes we will stick to observing response time as this connects more directly our experimental results to the schedulability tests.

6.2. Observed Response Time

Ideally, what we would want to know is the actual response time R_i of each mutator task as this would let us test schedulability. The response time is more important than the MMU as it accounts for all overheads and not just GC pauses. The observed response time accounts for the scheduler operations, quality of system timer, context switching cost, overheads of the virtual machine, mutator execution time, compiler inserted GC barriers and interrupts by the GC. We can measure experimentally the observed response time, which is defined as the duration between the time when a task becomes runnable and the completion of computation in that release. A periodic task becomes runnable at absolute times that are a multiple of the period plus the task's start time.

Figure 6 shows the response times of slack, periodic, and hybrid schedulers with the P workload, task 1 with a 10ms period. For each scheduler we provide a histogram of the

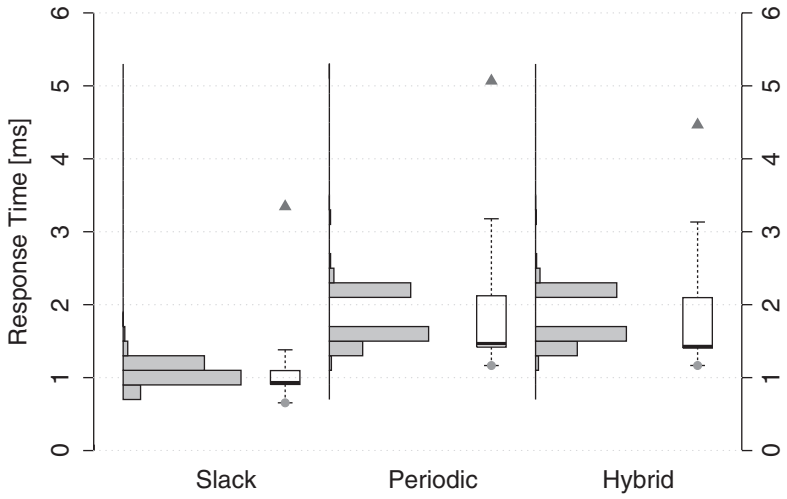


Fig. 6. *Observed Response Times.* Response times in ms for P workload.

measured times as well as box-plots (the bold central line marks the median, the hinges mark the quartiles, and the whiskers are each 1.5× the interquartile range from the closer quartile). We perform 50 executions of the benchmark (350,000 measurements). We report maximum times (red triangle) and minimum times (green bullet) over all executions, which characterize the true extremes of our system. The results show that slack scheduling gives better response times than periodic and hybrid scheduling, confirming our expectations. The distribution of response times with slack scheduling is narrower (has smaller inter-quartile range) than periodic and hybrid scheduling, which is again expected, given the periodic stealing of mutator quanta by the latter two. The overheads of periodic scheduling can be explained by the periodic scheduling algorithm. The 10ms window of the GC's schedule is aligned with the 10ms period of the task. Every task release that takes between 0.5ms and 1ms will be preempted by a single 0.5ms collector quantum. As the minimum response time measured with the slack scheduler is 0.7ms, the expected minimum for periodic is 1.2ms, which matches the measurements. The median response time with slack is 0.9ms and with periodic 1.5ms, which is again close to the expected value of 1.4ms. The minimum and median response times of hybrid are slightly smaller than that of periodic, which can be explained by more frequent cases when the collection is not in progress (hybrid uses slack for collection, and thus finishes it earlier than periodic). The maximum observed with slack was 3.3ms. This would mean 6 quanta (3ms) taken by the collector, thus a maximum of 6.3ms. However, the maximum of 5.1ms observed by periodic is smaller. This could be explained by a lack of GC work during the release that actually has the worst-case computation time. As the histograms show, the response time distribution has a long tail, and thus the long computation times are very unlikely; it probably happens that when they are triggered periodic collection is only active during part of the computation.

6.3. Schedulability Case Studies

This section mirrors Section 4.2 in our experimental setup. We use different RCD workloads to validate our conclusions about the respective power of the different approaches. We demonstrate that there are configurations that can be scheduled by a periodic GC and not by a slack-based one, and vice versa. We did not encounter a case where hybrid was beaten by either of the approaches in this workload.

We use two RCD workloads, P and S and artificially increase the amount of allocation, A_i, of the tasks to stress the memory subsystem. We run RCD with two tasks, such that task 2 does nothing else than create GC work and task 1 implements the collision detection algorithm. The results of this experiment are illustrated in Figure 7. The x-axis shows the number of objects allocated per release of task 2. The y-axis gives the highest observed response time for the corresponding amount of allocation. The end point of each line denotes the highest amount of allocation that can be generated without missing a deadline in the mutator or the GC.

Figure 7(a) shows the maximum observed times for both tasks with the RCD S workload. Task 1 has a mostly flat response time because the work performed each release is small and more or less constant. The figure shows that slack is faster than hybrid and periodic because task 1 is not preempted by the former. The maximum time of task 2 increases as we ask it to allocate more objects per release. Again, the difference between periodic and slack is explained by the fact that the periodic GC preempts both tasks and thus slows them down.

Figure 7(a) clearly shows that periodic "fails" by running out of memory earlier than either hybrid or slack. The slack and hybrid schedulers can sustain a larger level of allocation. Periodic-scheduled collection cannot keep up with the mutator because it does not make use of slack in the schedule. Even during idle time, it pauses periodically. Hybrid gives worse maximum response times than slack, because it also steals quanta from the mutator, in this case unnecessarily.[2]

Figure 7(b) is similar but uses the P workload and allocates millions of objects. In this example, the failures of periodic and hybrid are due to missed deadlines for the mutator tasks. In this configuration the periodic and hybrid schedulers can sustain a larger level of allocation. Task 2 has a period of 2s and allocates more memory than is available in the system. The collector must interrupt the task to avoid running out of memory. Slack-scheduled collection cannot do this, and thus the system crashes due to running out of memory. While the curves for hybrid and periodic schedulers overlap, the maximum response time is smaller for hybrid than for periodic. This is shown for task 1 in Figure 7(c). The figure also reveals an oscillation of the response times of slack and periodic within the range of 0.5 ms (for 1.2 to 2 millions of allocated objects). This is easily explained by the quantum length of 0.5 ms. A very small fluctuation in the maximum response results in different numbers of quanta stolen by the collector per mutator task release.

6.4. Relative GC Overheads

We have measured execution time overheads of the GC that we used for the comparison of slack and periodic scheduling over a nonincremental GC without any barriers. These overheads (Table II) quantify the cost in terms of execution time for turning a non-real-time GC into a real-time one. The percentage overhead $o(c, b)$ of GC configuration c running benchmark b is calculated as

$$o(c, b) = 100 \cdot \frac{\text{met}(c, b) - \text{met}(c_{\text{nonrt}}, b)}{\text{met}(c_{\text{nonrt}}, b)},$$

where $\text{met}(c, b)$ is mean execution time of benchmark b with GC configuration c over a number of iterations, selected on a per-benchmark basis, and configuration c_{nonrt} is of non-incremental GC without any barriers. The choice of arithmetic mean is common

[2]Anecdotally, in our experiments with Ovm, we observed rare but large pauses, about 40ms, that we could not explain at first. After careful study we found out that they were caused by the operating system discarding a memory page containing code. Reloading the page from disk generated the pause. Luckily, instructing the OS to lock all pages when they are first mapped fixed the problem.

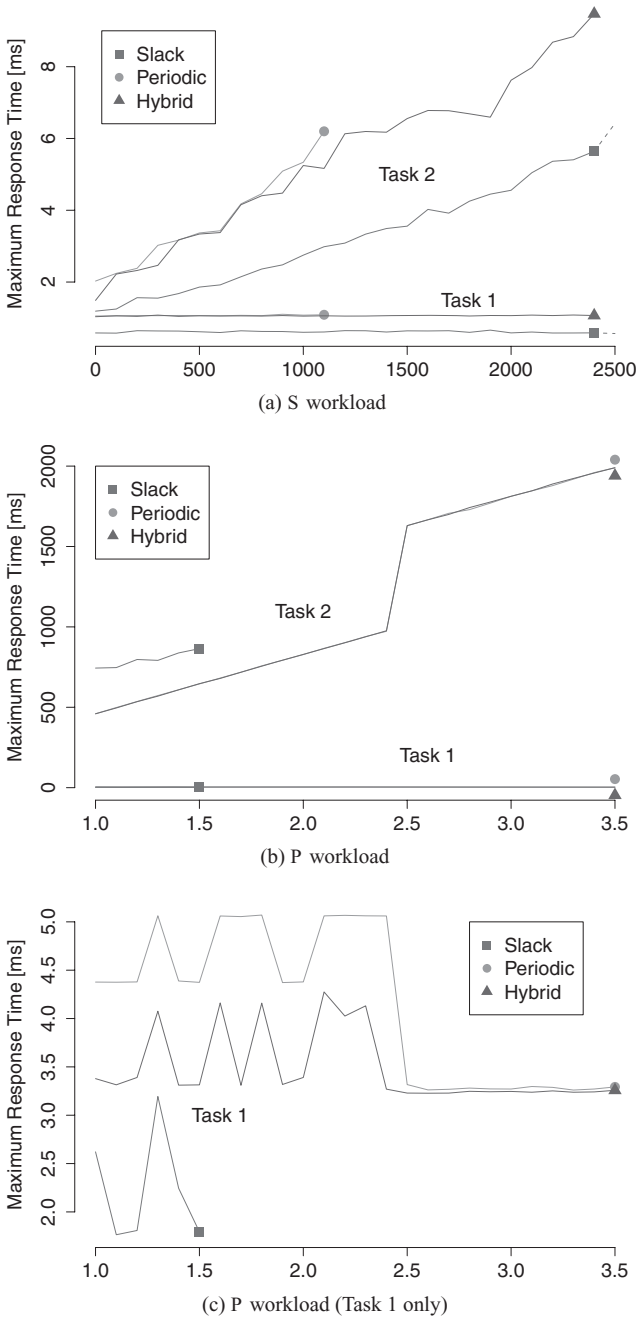


Fig. 7. *RCD workloads.* The x-axis is the number of objects allocated ((a) in thousands, (b) and (c) in millions) per release of Task 2. The y-axis gives the maximum observed response time (ms) of the tasks for different GC scheduling algorithms (lower is better).

Table II. Percentage Execution Time Overheads of a Real-Time GC (smaller is better)

	Non RT	Arraylets	Defr.	Defr.&Arr.
Antlr	−2	4	61	58
Bloat	27	40	114	121
Fop	1	5	38	39
Hsqldb	30	−4	41	30
Luindex	6	22	108	102
Lusearch	1	−15	47	27
Pmd	10	19	77	71
Xalan	−5	−2	25	20
Compress	−12	32	224	186
Jess	−6	3	62	66
Db	4	8	49	42
Javac	4	11	79	82
Mpegaudio	4	26	253	120
Mtrt	−14	−2	100	84
Jack	−6	7	60	62
Geo-Mean	2	9	81	**69**

and natural, as the sum of execution times has a physical meaning [Lilja 2000; Jain 1991]. The c_{nonrt} is a fair base as it does not include the feature of interest (feature that distinguished c from c_{nonrt}) and is conservative because relating to it leads to numerically larger relative overheads than if c_{nonrt} was in the numerator. We also include a geometric mean gmo(c) of the overheads over all benchmarks, which we calculate as

$$\text{gmo}(c) = 100 \cdot \exp\left(\frac{1}{\#_{\text{bench}}} \sum_{b} \ln\left(\frac{\text{met}(c,b)}{\text{met}(c_{\text{nonrt}},b)}\right)\right) - 100.$$

We use geometric mean on the grounds that we are looking for a relative overhead that is independent on the size of the workloads, but rather depends on their type [Jain 1991]. Geometric mean is quite robust to outliers and is used commonly, but the choice of a particular mean is not uncontroversial and some argue strongly against the use of geometric mean [Smith 1988; Lilja 2000]. We also show the individual benchmark overheads, allowing independent interpretation.

The results are obtained with selected benchmarks from the DaCapo [Blackburn et al. 2006a] and SPECjvm98 suites. Both suites include a diverse set of real and non-trivial applications from multiple domains, which are run on large data sets. Excluding standard libraries, the numbers of loaded methods are 152-1011 (SPEC) and 494-2433 (DaCapo) [Blackburn et al. 2006a]. The source code size of the DaCapo benchmarks is 850,000 lines of code (excluding libraries). Workload characterization of both SPEC and DaCapo benchmarks in respect to GC behavior has been provided in the DaCapo technical report [Blackburn et al. 2006b]. On average over all benchmarks (calculated by geometric mean), the execution time overhead of a full RTGC (defragmentation and arraylets) is 69%. The independent overhead of arraylets is 9% and the independent overhead of defragmentation is 81%. This means that arraylets, on average, cause a speedup when added to defragmentation, which is an interesting result, but not consistent over platforms. Our earlier measurements [Kalibera et al. 2009] with a different compiler and platform have shown joint overhead of 58%, independent overhead of arraylets 34%, and independent overhead of defragmentation of 40%. We have identified the compiler as a significant source of the difference. Now we use the recent version, GCC 4.4. We have verified that with GCC 4.1, which we used earlier [Kalibera et al. 2009b], arraylets do incur a slowdown even on the current platform: the joint overhead is 50%, the independent overhead of arraylets is 12%, and of defragmentation

Table III. Percentage of time spent in mutator (1) and percentages of time spent in different phases of the collector (2-6) as described in Section 3.1

	Collection Cycle Phase					
	Mutator	Collector				
	(1)	(2)	(3)	(4)	(5)	(6)
Antlr	94.3	0.2	38.7	0.2	55.4	5.6
Bloat	97.7	0.0	16.3	0.0	79.1	4.6
Fop	95.4	0.1	35.6	0.0	51.9	12.3
Hsqldb	87.1	0.0	71.5	0.0	22.4	6.1
Luindex	96.4	0.2	39.9	0.1	58.1	1.7
Lusearch	93.0	1.0	47.3	0.8	50.6	0.3
Pmd	91.5	0.1	24.6	0.0	72.8	2.5
Xalan	94.3	0.2	56.6	0.2	40.0	2.9
Compress	98.8	0.3	62.0	0.2	37.2	0.2
Jess	92.8	0.2	43.0	0.1	51.0	5.7
Db	96.6	0.1	71.1	0.1	28.2	0.4
Javac	93.2	0.0	33.7	0.0	49.9	16.3
Mtrt	87.2	0.2	55.3	0.2	41.1	3.4
Jack	91.4	0.2	34.7	0.1	55.0	9.9
Geo-Mean	93.5	0.0	41.9	0.0	47.1	2.8

(1) Wait Until Memory Is Scarce
(2) Scan Threads
(3) Mark And Clean Heap
(4) Clean Threads
(5) Sweep
(6) Defragment

46%. In our later work [Kalibera 2011], we identified the cause of the speed-up of arraylets when added to defragmentation with GCC 4.4—this version of the compiler did not inline many dereferences of Brooks forwarding pointers used for defragmentation, but it always inlined all dereferences to arraylets. Hence, with defragmentation, array accesses were faster with arraylets than without. We fixed the problem in [Kalibera 2011] by enforcing inlining of dereferences of Brooks forwarding pointers. With the fix, we obtained 8% independent overhead of arraylets, 10% independent overhead of defragmentation, and 16% joint overhead of arraylets and defragmentation (measured on another platform, but with the same compiler).

6.5. Relative Costs of GC Cycle Phases

We have measured the relative costs of different collection phases, presented in Table III. The first column refers to *Wait Until Memory is Scarce*, the number shown is the percentage of time spent in this phase relative to the whole program execution. On average using the geometric mean, 93.5% of time was spent in this phase, thus the collector only ran 6.5% of the total time. The percentages of time spent in individual collector phases are shown in the next columns, relative to the total time spent in the collector. The percentages are averaged using geometric mean from a number of iterations of each benchmark. These iterations are also from several executions of each benchmark to average out performance implications of non-deterministic initialization. The table shows that most of the collector time is spent in marking and sweeping of the heap (phases 3 and 5). The cost distribution between these two phases is, however, highly benchmark dependent. On average, 41.9% of time is spent in marking and 47.1% in sweeping. The relative costs of defragmentation also differ greatly among benchmarks. On average, 2.8% of collector time is spent in defragmentation. The costs of scanning and updating thread stacks (phases 2 and 4) are negligible, which is because of the use of pointer stacks [Baker et al. 2009].

7. RELATED WORK

Henriksson [1998] derived an initial schedulability analysis for a slack mostly concurrent two-space copying GC, which he then extended [Robertz and Henriksson 2003]. The core of the extended analysis applies, as shown in Schoeberl [2010], to any slack based single-heap GC, and thus we also use it in this paper. In this analysis, the GC is scheduled as a periodic task with lower priority than all other hard real-time

tasks. One period of the GC corresponds to one GC cycle. The analysis applies to non-defragmenting and certain defragmenting GCs, but it does not apply to our GC with defragmentation enabled. It is an open problem to find tractable memory bounds for Metronome-style defragmenting GC. However, the GC proposed by Henriksson [1998] has a significant weakness for hard real-time deployment. All live objects must be copied during collection, including large arrays. The GC must not prevent hard real-time tasks from running, and thus it can be interrupted even when copying. The GC must then restart the copy from the beginning because the mutator may have changed the original object. Frequent aborts slow down the GC, risking running out of memory. Moreover, GC progress is not guaranteed. This problem should either be addressed in schedulability analysis, proving that a system does have progress and does not run out of memory (based on the largest possible object that can be in the heap at any time), or in the implementation by avoiding the need for restarts or large memory copies. In other copying GCs, the need for restarts of copies is sometimes avoided by putting more responsibility on the mutator. In Baker [1978], a software barrier is proposed that redirects writes to the correct location, even if the object is being currently copied. The software solution however has a significant overhead [Schmidt and Nilsen 1994]. Smaller overheads were obtained by implementing the redirection using the memory hardware [Schmidt and Nilsen 1994]. A similar solution for copying GCs is always to access old copies of objects and store modifications to a mutation log, which can later be replayed by the collector [Nettles and O'Toole 1993].

Further extensions to schedulability analysis for a Henriksson-style copying GC assume that the GC is an aperiodic task running using a polling server at arbitrary fixed priority (not necessarily the lowest or highest priority in the system) [van Assche et al. 2006]. The traditional definition of the polling server is modified, so that an aperiodic task does not have to be ready at polling time to be serviced—it can get the (rest) of the server capacity even when it is ready after the polling time, but before the server runs out of capacity. The modified version of the polling server also differs from the deferrable server [Strosnider et al. 1995], because the server capacity is reduced while the server is waiting for an aperiodic task to be ready. This work focuses mainly on estimation of worst-case response times for an aperiodic task being run using a polling server, which is of general use for real-time systems. However, it does not address estimation of worst-case execution times for the GC in a GC cycle. Analyzing GC response time as an aperiodic task is itself already far more complex than the analysis of Robertz and Henriksson [2003]. The work is theoretical, with no tie to any existing GC implementation, though a version of the copying GC like Henriksson's is assumed. It explicitly ignores problems with re-starting object copying. Being scheduled by a polling server, GC scheduling is somewhat similar to periodic scheduling as used in existing GCs and as we describe it (if the polling server has the highest priority in the system). It also shows the advantage of periodic over slack scheduling, where a GC cycle being too long results in high memory requirements. However, a single polling server can only accurately describe a trivial interleaving of mutator and GC targeting 0.5 utilization, which is not used in today's periodic GCs. Modeling the GC as an aperiodic task as opposed to a periodic one is more realistic, though more complex.

Schedulability analysis for systems with RTGC has also been explored by Kim et al. [2001]. Again, a Henriksson-style copying two-space GC is assumed, with some proposed improvements. The GC work is modeled as an aperiodic task run using a sporadic server, at the highest priority in the system. Again, this model is similar to existing implementations of periodic scheduling for GC, but does not allow time schedules used by current periodic GCs. The work is evaluated using trace-driven simulation. It is unclear if the GC itself has been prototyped.

Metronome [Bacon et al. 2003b, 2003a] introduced periodic scheduling, based on the observation that it can absorb any unpredictable short-term allocation rate within the long-term allocation rate that is typically more predictable in most applications. Metronome used the Yuasa [1990] snapshot-at-the-beginning algorithm because it bounds collector work. Metronome uses arraylets to control external fragmentation and incremental defragmentation as a last defense against internal fragmentation, which is nevertheless rare because of Metronome's segregated free-list allocator. Metronome is the basis for IBM's WebSphere Real-Time product, which adds multiprocessor support, but lacks dynamic defragmentation. The published empirical evaluations of Metronome focus on verifying the intended low pause times and the MMU distribution. Although these are important for schedulability of systems with this type of GC, to our knowledge, we are the first to actually provide schedulability analysis together with memory requirement bounds that apply to Metronome.

Hybrid scheduling was introduced with Metronome-TS [Auerbach et al. 2008] which combined slack and periodic scheduling for multiprocessor systems, allowing the shifting of collector work to available processors, even in the presence of multiple real-time Java virtual machines running on the same system. It can use slack, if available in the system, but also can steal some mutator quanta as in periodic scheduling. Metronome-TS keeps track of the amount of slack time used by the collector, and also accounts for the nontrivial collector work performed by mutator threads. As long as the accumulated amount of GC work done during slack time is high enough, the collector refrains from stealing quanta from the mutator. This feature could provide better response times than our implementation of hybrid scheduling, though it remains an unanswered question how significant that improvement would be for worst-case time. Providing schedulability analysis for such a system remains an open research problem.

Siebert [1999] implemented another style of real-time GC for Java. His GC is work-based—there is no explicit GC thread to schedule—instead each allocation performed by the program performs a fixed quantity of collection work. Unfortunately, there has been no head-to-head comparison with time-triggered GCs. Conducting schedulability analysis of a work-based GC is straightforward: with no GC thread per se all the GC costs are imposed on the mutator threads. Their worst-case execution time increases proportionally to the number of operations they perform and traditional schedulability analysis can be performed to get response times. An additional test has to be added to ensure that work-based GC can reclaim memory fast enough to keep up with the mutator.

Generational GCs take advantage of the common generational behavior of applications, where young objects tend to die quickly, while older objects tend to survive several collections. These GCs collect two or more generations of objects independently, reducing the overhead of the collection, as the old generations do not have to be collected as often as the young ones. Even nonincremental generational GCs may thus have smaller pauses than nongenerational GCs. The problem for hard real-time, however, is that not all applications always display generational behavior, and thus small pauses would not be guaranteed. Still, incremental generational collection can be plugged into an RTGC to reduce the GC overhead for applications that do have generational behavior [Frampton et al. 2007]. Other applications, however, can suffer from slow-down and/or increased memory requirements [Frampton et al. 2007].

8. CONCLUSION

Real-time systems are designed to meet their deadlines and stay within their memory budget. GC must not get in the way of either requirement. Here, we have investigated the impact of scheduling policies on GC behavior. We have focused on time-triggered real-time GC algorithms and the two fundamental approaches to scheduling them:

periodic and slack-based scheduling. Both approaches are used in commercial products and are now being deployed in applications. We have developed schedulability tests for both approaches for fixed-priority scheduling and demonstrated that they have distinct limitations. In some cases, a system may be schedulable by only one of the scheduling policies. These results suggest that choosing the scheduling strategy is a key part of the design of real-time applications that use GC. Our implementation in the Minuteman framework on top of Ovm validates our results and shows that scheduling strategy matters. In order to present a fair comparison of both approaches we have reimplemented them independently. The two fundamental approaches are not entirely in conflict. A hybrid of time-based and slack-based scheduling has been proposed by others, yet not compared to the two original policies. We developed schedulability tests also for hybrid scheduling and implemented hybrid scheduling in Ovm. While hybrid scheduling can run systems that periodic or slack cannot, it does not eliminate the importance of policy selection: some systems can be run with slack, but not hybrid policy.

Our experimental results let us draw a number of conclusions. First, the minimum mutator utilization metric proposed in previous work as a way to characterize RTGC does not accurately depict the results of slack-based scheduling. Second, all application threads are affected by the GC design due to barriers inserted by the compiler on memory accesses. We have observed a mean of 69% slowdown on computational tasks. This overhead is 16% in a newer version of our virtual machine and could possibly be further reduced by additional compiler optimizations [Bacon et al. 2003b]. Still, some overhead will remain and has to be accounted for when designing a real-time application. Thirdly, our experimental results have demonstrated that neither of the basic scheduling strategies is strictly preferable. There will be applications than can be scheduled with a periodic GC and others with a slack-based GC. This also appears in our theoretical results. Overall hybrid scheduling performs best, but for any given application the optimal scheduling is highly dependent on the amount of the available slack and the rate of allocation.

ACKNOWLEDGMENTS

The authors thank David Bacon, Bertrand Delsart, Richard Jones, and Martin Schoeberl for their help and comments on this work.

REFERENCES

Armbruster, A., Baker, J., Cunei, A., Holmes, D., Flack, C., Pizlo, F., Pla, E., Prochazka, M., and Vitek, J. 2007. A real-time Java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst. 7,* 1, 1–49.

Auerbach, J., Bacon, D. F., Blainey, B., Cheng, P., Dawson, M., Fulton, M., Grove, D., Hart, D., and Stoodley, M. 2007. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software (EMSOFT)*. 249–258.

Auerbach, J., Bacon, D. F., Cheng, P., Grove, D., Biron, B., Gracie, C., McCloskey, B., Micic, A., and Sciampacone, R. 2008. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM/IEEE International Conference on Embedded Software (EMSOFT)*. 245–254.

Bacon, D. F., Cheng, P., and Rajan, V. T. 2003a. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 81–92.

Bacon, D. F., Cheng, P., and Rajan, V. T. 2003b. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 285–298.

Baker, H. G. 1978. List processing in real time on a serial computer. *Comm. of the ACM 21,* 4, 280–294.

Baker, J., Cunei, A., Kalibera, T., Pizlo, F., and Vitek, J. 2009. Accurate garbage collection in uncooperative environments revisited. *Concur. and Comput. Pract. Exper. 21,* 12, 1572–1606.

Blackburn, S., Garner, R., McKinley, K. S., Diwan, A., Guyer, S. Z., Hosking, A., Moss, J. E. B., Stefanović, D., et al. 2006a. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*. 169–190.

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A. L., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006b. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Tech. rep. TR-CS-06-01, Australian National University. http://www.dacapobench.org.

Bollella, G., Delsart, B., Guider, R., Lizzi, C., and Parain, F. 2005. Mackinac: Making HotSpot real-time. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. 45–54.

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

Brooks, R. A. 1984. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP)*. 256–262.

Burns, A., Tindell, K., and Wellings, A. 1995. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. Softw. Engine. 21,* 5, 475–480.

Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., Mathiske, B., and Wolczko, M. 2003. Heap compression for memory-constrained Java environments. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 282–301.

Cheng, P. and Blelloch, G. E. 2001. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 125–136.

Cheng, P. S.-D. 2001. Scalable real-time parallel garbage collection for symmetric multiprocessors. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Dijkstra, E., Lamport, L., Martin, A., Scholten, C., and Stefens, E. 1978. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM 21,* 11, 966–975.

Fidge, C. J. 1998. Real-time schedulability tests for preemptive multitasking. *Real-Time Syst. 14,* 1, 61–93.

Frampton, D., Bacon, D. F., Cheng, P., and Grove, D. 2007. Generational real-time garbage collection. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 101–125.

Garbervetsky, D., Yovine, S., Braberman, V. A., Rouaux, M., and Taboada, A. 2009. On transforming Java-like programs into memory-predictable code. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. 140–149.

Henriksson, R. 1998. Scheduling garbage collection in embedded systems. Ph.D. thesis, Lund University.

Honig Spring, J., Pizlo, F., Guerraoui, R., and Vitek, J. 2007. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments (VEE)*. 191–201.

Jain, R. 1991. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons.

Joseph, M. and Pandya, P. K. 1986. Finding response times in a real-time system. *Comput. J. 29,* 5, 390–395.

Kalibera, T. 2011. Replicating real-time garbage collector. *Concur. Comput.: Pract. Exper*. To appear. DOI: 10.1002/cpe.1669.

Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B., and Vitek, J. 2009a. CDx: A family of real-time Java benchmarks. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. 41–50.

Kalibera, T., Pizlo, F., Hosking, A. L., and Vitek, J. 2009b. Scheduling hard real-time garbage collection. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*. 81–92.

Kalibera, T., Prochazka, M., Pizlo, F., Decky, M., Vitek, J., and Zulianello, M. 2009c. Real-time Java in space: Potential benefits and open challenges. In *Proceedings of the Eurospace Conference on Data Systems in Aerospace (DASIA)*.

Kim, T., Chang, N., and Shin, H. 2001. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *J. Syst. Softw. 58,* 3, 247–260.

Lilja, D. J. 2000. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press.

Nettles, S. and O'Toole, J. 1993. Real-time replication garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. 217–226.

Pitter, C. and Schoeberl, M. 2010. A real-time java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst. 10,* 1, 9:1–9:34.

PIZLO, F. AND VITEK, J. 2008. Memory management for real-time Java: State of the art. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*.

PRINTEZIS, T. 2006. On measuring garbage collection responsiveness. *Sci. Comput. Program. 62,* 2, 164–183.

ROBERTZ, S. G. AND HENRIKSSON, R. 2003. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 93–102.

SCHMIDT, W. J. AND NILSEN, K. D. 1994. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the 6th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 76–85.

SCHOEBERL, M. 2010. Scheduling of hard real-time garbage collection. *Real-Time Syst. 45,* 3, 176–213.

SIEBERT, F. 1999. Real-time garbage collection in multi-threaded systems on a single processor. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*. 277–278.

SIEBERT, F. 2000. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 9–17.

SMITH, J. E. 1988. Characterizing computer performance with a single number. *Comm. ACM 31*, 1202–1206.

STROSNIDER, J. K., LEHOCZKY, J. P., AND SHA, L. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput. 44,* 1, 73–91.

VAN ASSCHE, M., GOOSSENS, J., AND DEVILLERS, R. R. 2006. Joint garbage collection and hard real-time scheduling. *J. Embed. Comput. 2,* 3–4, 313–326.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D. B., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P. P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst. 7,* 3, 36:1–36:53.

YUASA, T. 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw. 11,* 3, 181–198.

Received August 2010; revised March 2011; accepted May 2011