Scheduling Hard Real-time Garbage Collection

Tomas Kalibera, Filip Pizlo, Antony L. Hosking, Jan Vitek
Purdue University

***Abstract*—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 paper comprehensively compares the two leading approaches to hard real-time garbage collection. While there are many design decisions involved in selecting a real-time garbage collection algorithm, for time-based garbage collectors researchers and practitioners remain undecided as to whether to choose *periodic scheduling* or *slack-based scheduling*. A significant impediment to valid experimental comparison is that the commercial implementations use completely different proprietary infrastructures. Here, 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 two approaches, both experimentally using realistic workloads, and analytically in terms of schedulability.**

I. Introduction

Managed languages such as Java and C# are increasingly being considered for hard real-time applications. From both a technical and a scientific standpoint, 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 of time-triggered real-time garbage collection algorithms have been proposed in the literature and implemented in commercial products. This paper attempts to clarify some of the confusing 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 (links). 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, thread stacks, and 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 reachable objects are considered to be *live*: accessible at some time in the future by the application. Live objects may be defragmented.

In a real-time setting it is not practical to collect the heap atomically with respect to the mutator, since stopping the mutator threads may result in unacceptable GC pause times that risk missed deadlines. Thus, real-time GCs must work *incrementally*, interleaved with the normal execution of the real-time mutator tasks. Of course, this means that a real-time GC must cope safely with interleaved updates performed by the mutator tasks. 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 real-time GC (RTGC) and with support for the Real-time Specification for Java (RTSJ) [1], an extension to the Java programming language that is suitable for hard real-time applications as we have demonstrated in our previous work [2], [3], [4]. 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. This allows us to 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 controls [5].

The contributions of this paper are:

- **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 time-based RTGC algorithms including periodically scheduled [6] and slack-based [7] GCs.
- **Evaluation**: We empirically evaluate these alternative RTGC algorithms on a number of large standard benchmarks as well as on a real-time application.
- **Repeatability**: Our work is the first (and 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 (compiler, object layout, hardware platform, etc.) 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 RTGC algorithms.

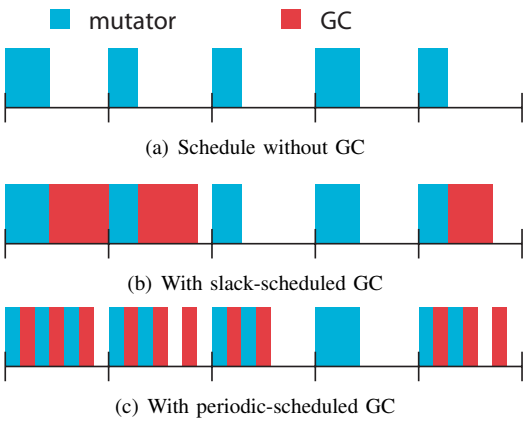
Links to implementation, benchmarks, and scripts can be found at http://www.ovmj.org/minuteman.

II. Real-Time Garbage Collection

The goal of a *real-time* GC 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 that developers be able to reason about the impact of selecting a particular GC algorithm on their application. The sources of space overhead for GC are mark bits (to record reachable objects), fragmentation due to segregated allocation, heap meta-data, and 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, reads and writes of references in the heap, and any other heap accesses.

Time predictability is often the main concern when selecting an RTGC. From the point of view of a real-time mutator 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 task, 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 alternatives: *slack-based* scheduling as first proposed by Henriksson [7], and adopted in the Sun Microsystems Java RTS product [8], and the *periodic* scheduling of Bacon, Cheng and Rajan [6] adopted in the IBM Websphere Real-time product.

A slack-based GC is implemented as 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) above. 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 must be enough "slack" in the schedule to leave time for the GC to satisfy the allocation operations performed by the application. 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) above).



(a) Schedule without GC

(b) With slack-scheduled GC

(c) With periodic-scheduled GC

Figure 1. Sample schedules for a periodic task

A periodic GC uses a different approach: the GC thread runs periodically with 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 (say 500μs) and the intervals are known *a priori*, it is possible to come up with an answer to questions (a) and (b). As for (c), there are also compiler-inserted barriers because the GC must be incremental. The difference between the approaches is illustrated graphically in Fig. 1. Note that there was no GC activity in the middle of the schedule, since all GC work was over. The following GC cycle started during the last period shown in the schedule.

While these approaches are not mutually exclusive (as shown by [9]), there has not been a clear comparison of the tradeoffs involved in selecting one over the other. The fact that the two production real-time environments for Java have implemented distinctly different alternatives with drastically different internal structure, makes benchmarks hard to compare.

III. 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. Thanks to restrictions placed on the reflective abilities of programs, we are able to statically compile the VM 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 [2]. Ovm implements most of the Real-time Specification for Java [1] and has been deployed on UAVs 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, periodic, and 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 (non-moving GC), or it can be enabled using Brooks forwarding pointers [10] or replication [11]. Barriers enabling incremental collection can be optimized either for throughput or for predictability. Arrays can be represented either using contiguous memory areas or split into displaced parts called *arraylets* [6].

For this study we select a GC that is a defragmenting, mostly non-copying, mark-and-sweep snapshot-at-the beginning Yuasa style [12] GC with weak tri-color invariant. The GC is fully incremental with defragmentation using Brooks forwarding pointers [10], barriers optimized for predictability (with minimum branches and activated at all times, even outside of the GC cycle), and arrays represented using arraylets. We build in support for periodic and slack scheduling, allowing us to compare the two in otherwise identical configurations. We provide more details below.

A. Collection Cycle

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

1. `waitUntilMemoryIsScarce`
2. `scanStacks`
3. `markAndCleanHeap`
4. `cleanStacks`
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. 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`*.

1. waitUntilMemoryIsScarce: At this stage, all existing objects are black, and new objects are allocated black. Any pointer can be dirty. The GC waits until the available memory falls below 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).

2. scanThreads: Start graph traversal. First, the values representing black and white are flipped, atomically making all objects in the heap white. The allocation color is now black. The GC's goal is to reclaim all unreachable objects that existed when the traversal started, while making sure no live object is reclaimed accidentally. This is achieved through the weak tri-color invariant enforced by the Yuasa barrier for the heap and the Dijkstra barrier for the stack. The weak tri-color invariant states that if a white object is pointed to by a black object, it must be also reachable from some other grey object through a chain of white nodes. The *Yuasa barrier* is added by the compiler to every store of a pointer into the heap. It captures the old pointer held at that location (before the store) and marks its target grey. This ensures that pointer stores do not remove a path from a grey object to a white object, regardless of whether the white object is actually referenced from a black object. The Yuasa barrier cannot be used for stack accesses. To prevent violating the weak tri-color invariant for references held in the stack, a Dijkstra barrier is added to every pointer store to a heap location. It marks (grey) the target of the new pointer being stored to that location. This ensures that no black object can ever point to a white object. The Dijkstra barrier makes scanning of stacks for pointers interruptible between threads but scanning a single thread is atomic. Luckily stack scanning is fast in Ovm due to the use of pointer stacks [13]. The target of each pointer in the stack is simply marked grey.

3. markAndCleanHeap: Grey objects are scanned one by one, marking all reachable objects grey, 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 location, so that after the whole heap is scanned, all pointers in the heap 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 (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 grey objects; global data is always assumed live. With the heap now clean, there are only white and black objects. White objects are garbage, black objects are assumed live, although some of them might have died since the GC cycle started.

4. cleanStacks: 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 location of moved objects. Fixing is again atomic with respect to each thread, but can be interrupted after each stack is fixed.

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 re-use. This involves some housekeeping of free memory and, since Java requires all allocated storage to be zeroed, also zeroing the freed memory. 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 re-use. 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 (2K). These pages can be used directly as arraylets or for large non-array objects, or they can be further partitioned for allocation of small objects. Large non-array 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 actually used by array data, the excess 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 per each page. The amount of this fragmentation versus the amount of object internal fragmentation can be controlled by tuning the number of supported object sizes. Moreover, it 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 re-used, 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 then 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.

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. 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 a size class for which most memory can be freed by defragmenting: moving objects from less occupied pages to more occupied pages. Each size class has a list of non-full pages, which is also used by the allocator. The defragmenter incrementally sorts this list in order of decreasing occupancy. Then it follows by copying 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. Finalization 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 re-usable for allocation during the sweep phase of the following GC cycle.

IV. Schedulability Analysis

We base our schedulability analysis on schedulability tests that allow modeling of periodic tasks in a synchronous system with any allocation of static priorities and without locking [14], [15]. Tasks are identified by integers, which are also their priorities, 1 (highest) to n. If C_i is the worst-case computation time per invocation of task i and T_i is the period and deadline, the maximum response time R_i occurs when all tasks are released for execution simultaneously with task i:

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

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

The recurrence is solved iteratively as follows:

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

The system is schedulable iff for every task i the iterative process converges to a finite fixed point R_i, such that $R_i < T_i$.

The intuitive explanation of (1) is that task i may need up to C_i time units to do the actual work, but can also be preempted by any higher priority task $j < i$. If preempted, the worst-case computation time of each higher priority task adds to task i's response time.

A. Schedulability and Garbage Collection

We assume the GC to be a periodic task with fixed period, T_{gc}. The period is chosen to be equal to the GC cycle which spans from the time GC activity starts until all unreachable objects have been freed. The work to be done by the GC depends on the memory operations performed by mutator tasks: each time the application allocates, loads from, or stores to heap pointer variables, work needs to be done by the GC. To capture the dependency between mutator actions and the GC, following Henriksson [7], we expect to know the worst-case amount of GC work, G_i (in time units), that a single invocation of a task i can generate. We also expect to have an upper bound to the "constant" GC work per cycle, G_0, which covers scanning of stacks and global variables (note that while this work isn't exactly constant, it is not dependent on the operation performed by the mutator). To capture the space requirements of GC, we need to know the size of the heap, H, and the maximum amount of live memory at any instant during any possible execution of the application, $L_{\max}$. We also need to know the maximum amount of allocation, A_i, per invocation of task i. Note that the GC period (GC cycle), T_{gc}, can be computed iteratively given the parameters (using the analysis we show later), such that the system is schedulable. The parameters are summarized in Tab. I.

The tests depend on two additional derived values: the maximum GC work $G_{\max}(T_{\text{gc}})$ and the maximum allocation $A_{\max}(T_{\text{gc}})$ per cycle.

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 (GC period)	system
G_0	[seconds]	GC cycle overhead	system

When T_{gc} is a multiple of the hyper-period ($\text{lcm}_{i=1..n}(T_i)$) and all tasks are started simultaneously, a simplified equation can be used to compute $G_{\max}(T_{\text{gc}})$ and $A_{\max}(T_{\text{gc}})$.

$$\begin{aligned} G_{\max}(T_{\text{gc}}) &= G_0 + \sum_{i=1}^{n} \left(\left\lceil \frac{T_{\text{gc}}}{T_i} \right\rceil G_i \right) \\ A_{\max}(T_{\text{gc}}) &= \sum_{i=1}^{n} \left(\left\lceil \frac{T_{\text{gc}}}{T_i} \right\rceil A_i \right) \end{aligned}$$

This equation can lead to longer GC cycles (in order to align them to the hyperperiod) and thus higher memory requirements. In general, the estimates must take into account the fact that we do not know how the generation of allocation/GC work is distributed within the execution of each task. We thus have to assume the highest possible number of (potentially partial) task invocations during a GC cycle, where all the GC work is bunched. The general equations are:

$$G_{\max}(T_{\text{gc}}) = G_0 + \sum_{i=1}^{n} \left(\left\lceil \frac{T_{\text{gc}}}{T_i} \right\rceil + 1 \right) G_i \quad (2)$$

$$A_{\max}(T_{\text{gc}}) = \sum_{i=1}^{n} \left(\left\lceil \frac{T_{\text{gc}}}{T_i} \right\rceil + 1 \right) A_i \quad (3)$$

In summary, there are three tests for a system with GC:

- TST-1: mutator tasks meet their deadlines;
- TST-2: GC meets its deadline (keeps up with tasks that use memory);
- TST-3: the system will not run out of memory with the given heap size.

Only TST-3 can be formulated independently of GC scheduling. We can find an upper bound, T_{gc}, on the GC cycle duration (GC period) that ensures that all allocation requests during a GC cycle can be fulfilled. We use a bound based on Robertz and Henriksson's [16], which we refine to the general case where T_{gc} is not a multiple of the hyper-period:

$$A_{\max}(T_{\text{gc}}) \leq \frac{H - L_{\max}}{2} \quad (4)$$

It is easy to see that the condition is necessary for the system not to run out of memory in the way captured by TST-3, because memory allocated (black) during a cycle, *floating garbage*, can only be freed by the end of the following cycle. Robertz and Henrikkson [16] and Schoeberl [17] show that this is also a sufficient condition. The intuition behind the proofs is as follows. The worst case is when the amount of live memory is $L_{\max}$ for all cycles. Also, the maximum amount of floating garbage G_{max} is the maximum allocatable memory in the previous cycle ($G_{max} = A_{\max}(T_{\text{gc}})$). The system does not run out of memory as long as $G_{max} + A_{\max}(T_{\text{gc}}) + L_{\max} = H$, and thus (4) is sufficient.

1) Slack GC: Mutator task deadlines, TST-1, are checked as if there was no GC, because the GC has the lowest priority in the system, and thus does not influence response times of the mutator tasks. The GC response time, TST-2, can be calculated as follows:

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

This recurrence is indeed very similar to the tests for mutator periodic real-time tasks: $G_{\max}(T_{\text{gc}})$ is the maximum computation time during the GC cycle and the second term denotes that the GC can be preempted by any mutator task in the system. The iterative process to find R_{gc} is also similar, except that here we can safely start with $R_{\text{gc}}^{(0)} := \sum_{i=1}^{n} C_i$.

2) Periodic GC: A periodic GC is scheduled based on a *time schedule*, independently on the mutator tasks, preempting them if necessary. The schedule thus impacts both TST-1 and TST-2.

The schedule is designed such that GC pause times are small (e.g., 500 μs) and that over a long time interval, the minimum mutator utilization (MMU) [18] is at least a given *target utilization* (e.g., u=0.7). To bound the pause times, the schedule never allows the GC to run for more than a single *time quantum* (= maximum GC pause time). To bound the utilization, the schedule ensures that from each *time window*[2] (e.g., 10ms - 20 time quanta), the fraction of time quanta for the mutator is at least u (e.g., 14 quanta for u=0.7). To achieve these goals, the schedule is a sequence formed by a repeating pattern (e.g., `MCMCMCMCMCMCMMMMMMMM`). If a GC does not have work to do, the mutator runs, but not vice-versa – the GC cannot use slack.

We formulate the schedulability analysis using functions $\text{mmu}(t)$ and $\text{mcu}(t)$, which, based on the schedule, give the minimum utilization [18] of mutator and GC, respectively, for an interval of length t. These functions are defined by the time schedule and can be computed accordingly by a program, but it is unlikely that they could be expressed precisely by a nice closed-form formula. For large t compared to the window size, we can however approximate these functions using target utilization u only, ignoring the schedule, as follows: $\text{mmu}(t) = 1 - \text{mcu}(t) = u$.

The task response time depends on mmu and on the maximum amount of work per cycle (for TST-1):

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

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

The GC cycle worst-case response time, TST-2, can be calculated as:

$$R_{\text{gc}} = \frac{1}{\text{mcu}(R_{\text{gc}})} G_{\max}(T_{\text{gc}}) \tag{7}$$

The iterative process starts with R_{gc}^{0} such that $\text{mcu}(R_{\text{gc}}) > 0$ (i.e., the periodic scheduler window size).

B. Examples

We illustrate the use of these schedulability equations with two extreme cases. The first example, Tab. II, is adapted from [7].

Table II
INPUT PARAMETERS FOR EXAMPLE "SLACK WINS"

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

Let's check if the example is schedulable. First, we check if mutator tasks always meet their deadlines (this is still classical scheduling theory [14]). For this we need C_is and T_is from Tab. II. The iterative solution of (1) gives the response times and reveals the mutator tasks are schedulable:

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

Next, we test if GC can keep up with the mutator. In addition to C_i and T_i, we also need $G_{\max}(T_{\text{gc}})$, which in turn requires G_0, G_i, and T_{gc}. From (2) we get $G_{\max}(730) = 10 + 75 \cdot 1 + 16 \cdot 5 + 9 \cdot 4 = 201$. Solving (5) we get $R_{\text{gc}} = 720 < 730 = T_{\text{gc}}$, which means that the GC can keep up with mutator. Last, we must check that the system does not run out of memory. For this we need T_is, A_is, H, T_{gc} and $L_{\max}$. From (2) we get $A_{\max}(730) = 12536$, which satisfies (4): $12536 < 12600$. The system thus does not run out of memory.

The example is however not schedulable with periodic GC as task 3 misses its deadline.

Now, consider the example of Tab. III with a periodic GC. Solving (6) gives task response times of 13 and 927. Mutator tasks are thus schedulable. The upper bound for T_{gc} is 149 given the available heap size. The system thus will not run out memory during the GC cycle. For TST-2 we need to solve (7): $R_{\text{gc}} = 127$, which is well within the GC cycle length. Indeed, the slack GC could never make the GC deadline, as the GC cycle is shorter than the period of one

[2]We never use the term *period* for time window to avoid confusion with task periods.

of the tasks. This example is thus not schedulable with a slack GC.

Table III
INPUT PARAMETERS FOR EXAMPLE "PERIODIC WINS"

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

V. EMPIRICAL EVALUATION SETTINGS

We prepared our empirical evaluation so that we can compare periodic and slack scheduling in Minuteman. To show that our GC is roughly comparable to the state-of-the art implementations, we also measured the periodic real-time GC implemented in the IBM WebSphere Real-Time Java Virtual Machine (J9) and the slack GC of the Sun Microsystems Java Real-Time System (RTS). Although RTS implements much more complex RTGC algorithm than slack scheduling [19], it effectively behaves like slack scheduling for hard real-time threads and the experiments we ran. We ran our experiments on a uniprocessor Intel Pentium 4 machine with Linux 2.6 patched for real-time.

A. Real-Time Collision Detector

For the empirical evaluation, we use our Real-Time Collision Detector (RCD) benchmark [20]. 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 know 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 2-d quadrants containing multiple aircraft, while in the second stage it performs 3-d 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).

The benchmark reports response times of the collision detector, which are measured from idealized release times (no jitter, ideal timer). The idealized release times are defined by absolute start time, which is rounded to avoid phase-shift of real timers [21], and by the period. We use a 10 ms period, which matches the periodic scheduler window size.

We also use two different workloads, M and V. We dimension the M workload so that Ovm, J9, and RTS can all successfully run it. It has fewer aircraft (19), no collisions, no background noise – only the detector task is running during the measurement – and a very large heap. A smaller heap size however causes one of the production systems to run out of memory. For this workload, we disable defragmentation in Ovm to be closer to the production systems.

The V workload is designed to be more challenging: it has 80 aircraft with actual collisions, runs the SPECjvm98 javac benchmark as background noise, and has a smaller heap. Moreover, the javac benchmark run in the background triggers defragmentation, which is enabled in Ovm for this workload.

To make the benchmark reasonably challenging for GC, we add a simple allocation noise generator into the real-time detector task. The noise generator allocates a given fixed number (i.e., 5000) of 64-byte objects per task invocation. Pointers to the objects are stored into a cyclic buffer for 1 million objects, so that over 2000 invocations, 5000 of these objects die at each invocation. To focus on steady state performance, we skip the first 2000 invocations. We evaluate data from 8000 invocations (corresponding to about 80s of wall clock time). To average out start-up non-determinism, we repeat executions (the numbers of repetitions differ for each experiment, as explained later).

In the V workload, we increase the amount of GC work generated even more. Instead of a constant object size, the allocations follow a predefined pattern, parameterized by maximum allocation request size and an allocation size increment.

B. GC Scheduler Configuration

The periodic schedulers (both Ovm and J9) use a 10ms window, 500μs quanta, a 1ms maximum pause time, and $u = 0.7$ target MMU. The slack schedulers (both Ovm and RTS) are configured so that GC runs at a lower priority than the real-time detector task. For the V workload (Ovm only), we run the slack scheduler with a priority above the background noise. Without this the slack scheduler would ultimately run out of memory, because the background (non-realtime) noise does not have enough slack. This necessarily favors the slack scheduler over periodic.

VI. EMPIRICAL EVALUATION RESULTS

We compare periodic and slack scheduling using three different metrics: MMU, maximum response time, and maximum amount of allocation noise that can be injected reliably.

A. Minimum Mutator Utilization

Target MMU is a parameter that controls the behavior of the periodic scheduler. Originally, it was proposed [18] as a metric that characterizes the quality of real-time GC. Although MMU can be used to bound worst-case response times and prove that deadlines cannot be missed, as we have shown for the periodic GC, a GC with better MMU does not necessarily perform better in a hard real-time setting.

This fact is demonstrated in Fig. 2. The MMU of our implementation of the periodic scheduler converges to the ideal case and target utilization of 0.7. Conversely, the slack scheduler has very poor MMU, with maximum pause time almost 10ms. The key problem of MMU as an evaluation metric for hard real-time is that it cannot capture that the almost 10ms pause time happens only when the mutator cannot run, and thus does not interfere with task response times.

Another general problem of MMU, if based on when mutator or GC threads are running, is that it puts an imprecise boundary between mutator and GC. It charges allocation costs and costs of memory-related operations (i.e., heap and stack accesses and their associated barriers) to the mutator. However, these costs are a major feature of the GC design with clear tradeoffs: they can be distributed among mutator and GC tasks to a large extent. Thus, MMU is not good for comparing different GCs because the costs of those GCs charged to MMU are part of the GC design. However, this is not an issue in our experiments with Minuteman, which allows us to change only the scheduling of collection, while leaving the barrier costs the same, so that the mutator/GC overheads remain fixed. Finer-grained accounting of mutator and GC activity has recently been introduced with the Metronome-TS GC [9].

B. Response Time

The response time is a much more suitable metric for hard real-time performance. Fig. 3 shows the response times of slack and periodic schedulers with the M workload. For each scheduler, we provide a histogram of the measured times, as well as box-plots. We perform 50 executions of the benchmark (400,000 measurements), recording maximum times (red triangle) and minimum times (green bullet) over all 50 executions, which characterize the true extremes on our system.

The results show that periodic scheduling with Ovm gives about the same best response times as slack scheduling, but worse median as well as maximum response times. Note that the overhead in response times can be easily explained by the periodic GC schedule. From the results of slack scheduling, the mutator runs up to between 4 and 4.5 ms per detector release, which is nine 500μs quanta. According to the periodic GC schedule, these 9 mutator quanta will be interleaved by 6 GC quanta, slowing down the mutator by 3 ms. Hence, the maximum response time for periodic scheduling is between 7 and 7.5 ms, which matches the results in the figure. Similarly, in the median case, the mutator needs 6 quanta, thus will be delayed by 4 GC quanta (2 ms), hence the median response time for periodic scheduling is about 2ms longer than for slack scheduling. In the best case, the GC is not needed (the system waits for next GC cycle to start), and thus response times of both periodic and slack are the same.

Ovm is within reasonable range of the performance of the product implementations. These results should however not be used for direct comparison of the two production systems, as we did not (and could not) make sure that they were configured optimally and fairly for such comparison. For the same reason, the comparison of the products to Ovm is only approximate.

Fig. 4 shows the response times of the slack and periodic schedulers in Ovm with the V workload. Note that the overhead of periodic can again be explained by the schedule of the periodic scheduler. From the slack numbers, we know that the GC takes about 11 quanta maximum (6 is median) per task invocation. A periodic scheduler would thus give 6 quanta (5 median) to the GC, creating a 3ms (2.5 ms median) overhead, which matches the measured results. The minimum response times are almost equal, which can be easily explained by the lack of GC activity during the best-case detector releases. The production VMs could failed on this workload.

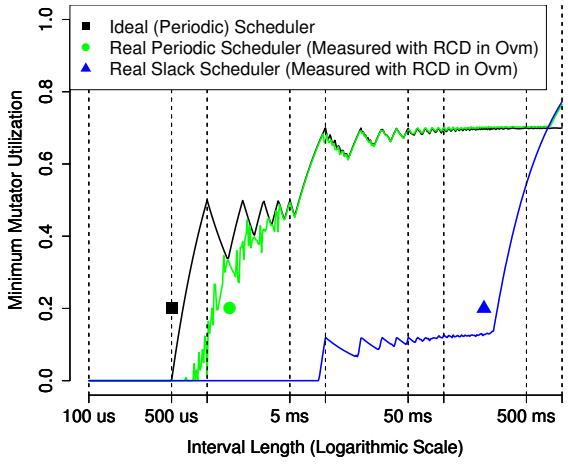


Figure 2. Minimum Mutator Utilization

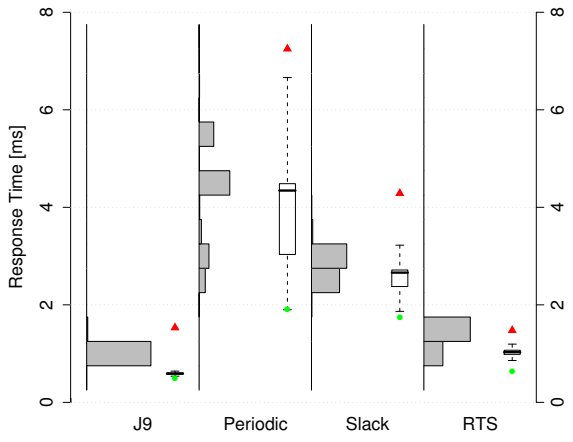


Figure 3. Response times for M workload

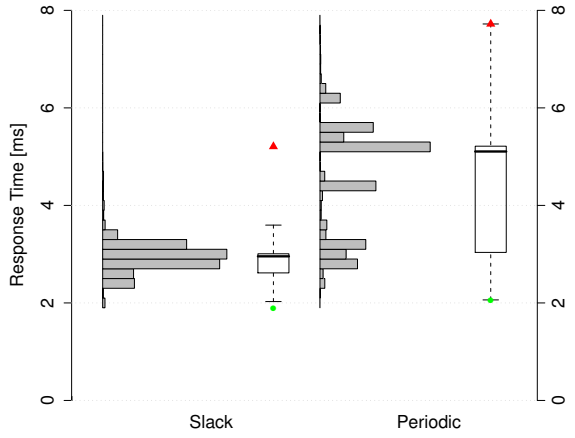


Figure 4. Response times for V workload

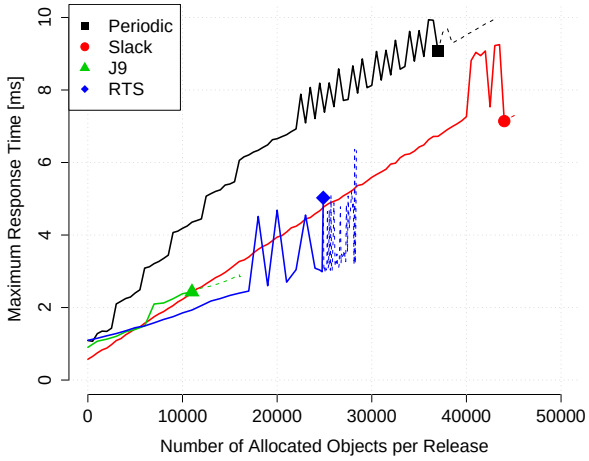


Figure 5. Maximum response with noise for M workload

C. Stress Test

To compare the robustness of each scheduler, we have run tests with different amounts of noise generated in the detector thread (Fig. 5 and Fig. 6). The magnified data points denote the highest amount of noise when all executions have succeeded (i.e., not missed a deadline and not run out of memory). With the M workload, Ovm periodic was only failing by missing deadlines and Ovm slack only by running out of memory. J9 and RTS were failing mostly by running out of memory, but deadline misses were also present.

With the more memory intensive V workload, all observed failures were running out of memory. However, neither J9 nor RTS could run this workload even for the smallest stress level. Moreover, RTS could not be configured for pure slack scheduling with the background noise (due to memory reservation and priority boosting features). Still, even with these features, RTS could not run even with the lowest stress level.

The results show that slack schedulers are more robust than periodic schedulers, and also that Ovm is more robust to increased allocation noise than the products. Slack schedulers have better maximum response times than periodic and the products have better maximum response times than Ovm.

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, instruct the OS to lock all pages when they are first mapped fixed the problem.

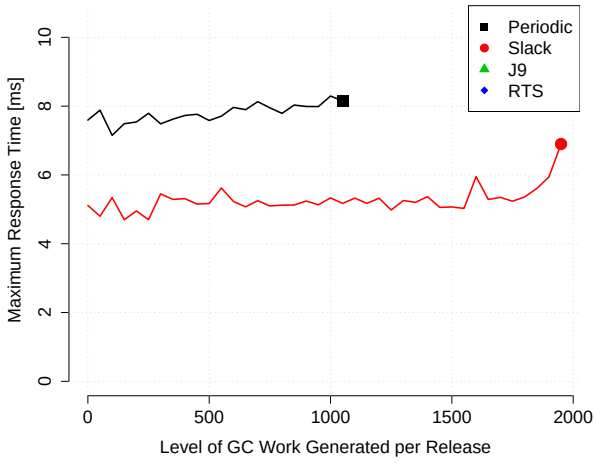


Figure 6. Maximum response with noise for V workload

D. Relative GC Overheads

We have measured overheads of the GC that we used for the comparison of slack and periodic scheduling over a non-incremental GC without any barriers. These overheads (Table IV) quantify the price 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

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

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. We also include a geometric mean $\text{gmo}(c)$ of the overheads over all benchmarks, which we calculate as

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

The results are obtained with selected benchmarks from the Dacapo 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 [22]. Excluding

Table IV
PERCENTAGE OVERHEADS OF A REAL-TIME GC (SMALLER IS BETTER)

	Base RT	Arraylets	Defr.	Defr.&Arr.
Antlr	1	21	40	58
Bloat	25	51	140	112
Fop	3	16	34	43
Hsqldb	27	59	79	89
Luindex	8	27	44	66
Lusearch	1	-2	41	33
Pmd	16	30	43	52
Xalan	-1	2	12	14
Compress	-1	42	44	60
Db	11	26	16	30
Jack	6	52	24	68
Javac	19	65	39	93
Jess	13	66	22	73
Mpegaudio	4	48	54	82
Mtrt	3	24	12	30
Geo-Mean	9	34	40	**58**

standard libraries, the numbers of loaded methods are 152-1011 (SPEC) and 494-2433 (Dacapo) [22]. 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 [22].

On geometric average over all benchmarks, the overhead of a full RTGC (defragmentation and arraylets) is 58%. Independent overheads of arraylets and defragmentation are both similar and still relatively high, 34% and 40%. The overhead of incrementality alone is much smaller, only 9%. The results also suggest that the overheads are strongly benchmark dependent.

VII. RELATED WORK

Henriksson derived an initial schedulability analysis for a slack mostly-concurrent two-space copying GC [7], which he then extended [16]. The core of the extended analysis applies, as shown in [17], 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-compacting and certain compacting GCs, however it does not apply to our GC with compaction enabled. It is an outstanding problem to find tractable memory bounds for Metronome-style compacting GC.

The GC proposed by Henrikkson [7] however 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 [23], 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 [24]. Smaller overheads were obtained by implementing the redirection using the memory hardware [24]. A similar solution for copying GCs is always to access old copies of objects and store modifications to a mutation log, which can be later replayed by the collector [11]. This nicely integrates into generational collectors, because they already have mutation logs.

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) [25]. 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 [26], 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. Analysing GC response time as an aperiodic task is itself already far more complex than Henriksson's analysis [16].

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 drastically more complex.

Schedulability analysis for systems with RTGC has also been explored by Kim et al. [27]. 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 [6], [28] is the implementation of periodic GC in IBM's WebSphere Real-Time (J9) product. Although the original prototype implementation of Metronome in an older version of Jikes RVM had defragmentation, the current implementation does not and may suffer from fragmentation.

The published empirical evaluations of Metronome focus on verifying the intended low pause times and the MMU distribution. Although these are indeed 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.

Metronome-TS [9] combines slack and periodic scheduling. It can use slack, if available in the system, but also can steal some mutator quanta as in periodic scheduling. When enough slack GC work has been done to make sure the system will not run out of memory, the GC does steal quanta from the mutator, thus improving the response times of real-time tasks. This hybrid can perform better than both slack and periodic, but a schedulability analysis that could provably show this remains an interesting outstanding problem.

Siebert [29] implemented another style of real-time GC for Java. His GC is work-based – there is no explicit GC thread to schedule – instead each operation performed by the program (read, write, allocate) does a fixed quantity of collection work. While this style of GC has been dismissed by some as less efficient, there has been no head-to-head comparison with time-triggered GCs.

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 non-incremental generational GCs may thus have smaller pauses than non-generational GCs. The problem for hard real-time, however, is that not all applications have always the generational behavior, and thus small pauses would not be granted. Still, incremental generational collection can be plugged into an RTGC to reduce the GC overhead for applications that do have generational behavior [30]. Other applications, however, can suffer from slow-down and/or increased memory requirements [30].

VIII. Conclusion

Hard 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: time-based 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 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 does indeed matter. In order to present a fair comparison of both approaches we have reimplemented them independently. We have compared the performance of our framework with commercial Java VMs to ensure that ours is representative.

Our experimental results let us draw a number of important 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 58% slowdown in throughput (and up to 112%) on computational tasks. We believe that with additional compiler optimizations this overhead may be reduced significantly, yet some overhead will always remain and must be accounted for when designing a real-time application. We have used a sample real-time benchmark with two workloads (one fairly light and the other including background computational and allocation noise from a non-real-time task). Response time results on this particular benchmark favor slack-based scheduling, and show that our implementation is representative of the performance of commercial systems.

Acknowledgements

The authors thanks David Bacon, Bertrand Delsart and Martin Schoeberl for their help and comments on this work. This work is supported by the National Science Foundation under grants Nos. CCF-0702240, and CCF-0811691, and by Microsoft, Intel, and IBM. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors. This research partially done while Hosking on leave as Visiting Fellow, Department of Computer Science, Australian National University.

REFERENCES

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

[2] A. Armbruster, J. Baker, A. Cunei, D. Holmes, C. Flack, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, "A Real-time Java virtual machine with applications in avionics," *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 7, no. 1, 2007.

[3] F. Pizlo and J. Vitek, "Memory management for real-time Java: State of the art," in *Proceedings of the IEEE Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, 2008.

[4] J. Honig Spring, F. Pizlo, R. Guerraoui, and J. Vitek, "Reflexes: Abstractions for highly responsive systems," in *Proceedings of the ACM/USENIX Conference on Virtual Execution Environments (VEE)*, 2007.

[5] T. Kalibera, M. Prochazka, F. Pizlo, M. Decky, J. Vitek, and M. Zulianello, "Real-time Java in space: Potential benefits and open challenges," in *Proceedings of Data Systems in Aerospace (DASIA)*, 2009.

[6] D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[7] R. Henriksson, "Scheduling garbage collection in embedded systems," Ph.D. dissertation, Lund University, 1998.

[8] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain, "Mackinac: Making hotspot real-time," in *Proceedings of the Eighth IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2005.

[9] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone, "Tax-and-spend: democratic scheduling for real-time garbage collection," in *Proceedings of the ACM Conference on Embedded Software (EMSOFT)*, 2008.

[10] R. A. Brooks, "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)*, 1984.

[11] S. Nettles and J. O'Toole, "Real-time replication garbage collection," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[12] T. Yuasa, "Real-time garbage collection on general-purpose machines," *Journal of Systems and Software*, vol. 11, no. 3, 1990.

[13] J. Baker, A. Cunei, F. Pizlo, and J. Vitek, "Accurate garbage collection in uncooperative environments," *Concurrency and Computation: Practice and Experience*, 2009.

[14] M. Joseph and P. K. Pandya, "Finding response times in a real-time system," *Computer Journal*, vol. 29, no. 5, 1986.

[15] C. J. Fidge, "Real-time schedulability tests for preemptive multitasking," *Real-Time Systems*, vol. 14, no. 1, 1998.

[16] S. Gestegard Robertz and R. Henriksson, "Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems," in *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003.

[17] M. Schoeberl, "Real-time garbage collection for Java," in *IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2006.

[18] P. Cheng and G. E. Blelloch, "A parallel, real-time garbage collector," in *Proceedings of the ACM Conference on Programming language design and implementation (PLDI)*, 2001.

[19] E. J. Bruno and G. Bollella, *Real-Time Java Programming with Java RTS*. Prentice Hall, 2009.

[20] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek, "CDx: A family of real-time Java benchmarks," in *Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.

[21] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 21, no. 5, 1995.

[22] S. M. Blackburn, *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the ACM Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2006.

[23] H. G. Baker, "List processing in real time on a serial computer," *Communications of the ACM*, vol. 21, no. 4, Apr. 1978.

[24] W. J. Schmidt and K. D. Nilsen, "Performance of a hardware-assisted real-time garbage collector," in *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[25] M. van Assche, J. Goossens, and R. Devillers, "Joint garbage collection and hard real-time scheduling," *Journal on Embedded Computing*, vol. 2, no. 3,4, 2006.

[26] J. Strosnider, J. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, 1995.

[27] T. Kim, N. Chang, and H. Shin, "Joint scheduling of garbage collector and hard real-time tasks for embedded applications," *Journal of Systems and Software*, vol. 58, no. 3, 2001.

[28] D. F. Bacon, P. Cheng, and V. T. Rajan, "Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java," in *Proceedings of the ACM Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, 2003.

[29] F. Siebert, "Real-time garbage collection in multi-threaded systems on a single processor," in *IEEE Real-Time Systems Symposium (RTSS'99)*, 1999.

[30] D. Frampton, D. F. Bacon, P. Cheng, and D. Grove, "Generational real-time garbage collection," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2007.