Tax-and-Spend: Democratic Scheduling for Real-time Garbage Collection

Joshua Auerbach
IBM Research
josh@us.ibm.com

David F. Bacon
IBM Research
dfb@watson.ibm.com

Perry Cheng
IBM Research
perryisreading@gmail.com

David Grove
IBM Research
groved@us.ibm.com

Ben Biron
IBM Software Group
benbiron@ca.ibm.com

Charlie Gracie
IBM Software Group
cgracie@ca.ibm.com

Bill McCloskey
U.C. Berkeley
billm@cs.berkeley.edu

Aleksandar Micic
IBM Software Group
amicic@ca.ibm.com

Ryan Sciampacone
IBM Software Group
rsciampa@ca.ibm.com

ABSTRACT

Real-time Garbage Collection (RTGC) has recently advanced to the point where it is being used in production for financial trading, military command-and-control, and telecommunications. However, among potential users of RTGC, there is enormous diversity in both application requirements and deployment environments.

Previously described RTGCs tend to work well in a narrow band of possible environments, leading to fragile systems and limiting adoption of real-time garbage collection technology.

This paper introduces a collector scheduling methodology called *tax-and-spend* and the collector design revisions needed to support it. Tax-and-spend provides a general *mechanism* which works well across a variety of application, machine, and operating system configurations. Tax-and-spend subsumes the predominant pre-existing RTGC scheduling techniques. It allows different *policies* to be applied in different contexts depending on the needs of the application. Virtual machines can co-exist compositionally on a single machine.

We describe the implementation of our system, Metronome-TS, as an extension of the Metronome collector in IBM's Real-time J9 virtual machine product, and we evaluate it running on an 8-way SMP blade with a real-time Linux kernel. Compared to the state-of-the-art Metronome system on which it is based, implemented in the identical infrastructure, it achieves almost 3x shorter latencies, comparable utilization at a 2.5x shorter time window, and mean throughput improvements of 10-20%.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Experimentation, Languages, Performance

Keywords

Java, JVM, Garbage Collection, Real Time

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

EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.

Copyright 2008 ACM 978-1-60558-468-3/08/10 ...$5.00.

1. INTRODUCTION

Recent advances in real-time garbage collection have led to its production use in a number of significant application areas, including financial trading, telecommunications, and military command-and-control. For instance, IBM's Metronome collector [4] is being used in the Navy's Zumwalt-class destroyer.

However, there is a very wide variety of users interested in RTGC technology, with a wide variety of application characteristics and operating environments. Real-time applications may be periodic (e.g. avionics), queued (e.g. telecommunication switches), or interactive (e.g. video games).

Any of these systems may be either "hard" or "soft" real-time. We have built both helicopter flight control (generally considered hard real-time) [3] and music synthesis (generally considered soft real-time) [2] systems as classical periodic real-time systems. Similarly, we have worked with customers building both telecommunications and missile-defense systems as queued real-time systems.

Classical real-time systems typically operate with a certain amount of *slack*. If there is no slack, then the system can't guarantee to meet its bounds. But interactive systems may become sluggish during bursts of other activity on the machine, and in some cases this is the desired behavior. Queued systems may absorb temporary load spikes and tolerate a certain fraction of outliers depending on a service-level agreement. And the use of adaptive algorithms (that compute successively more precise results until available slack is exhausted) allows hard real-time systems to saturate the CPU.

Furthermore, there is a great deal of diversity in the operating environment: the system may be a *uni- or multi-processor*; the machine may be *dedicated or shared*; and the operating system may have *varying levels of real-time support*.

Currently, RTGCs only support a limited subset of the potential application types and operating environments. For instance, Metronome works best on dedicated uni- or small multi-processor systems, and it requires a variety of special RTOS features. BEA's soft-real-time collector is more widely deployable, but it suffers latencies as much as 100 times worse. Henriksson's collector [15]

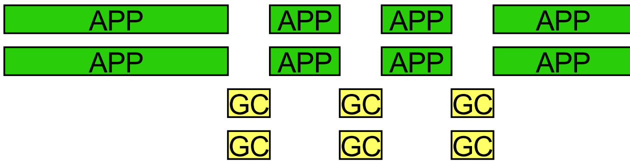


Figure 1: Mutator and GC interleaving in Metronome. All mutators are paused when the GC runs.

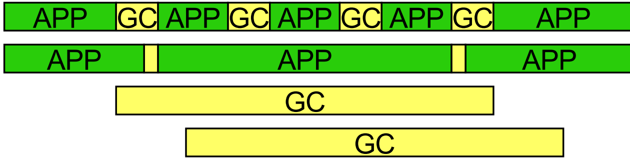


Figure 2: Mutator and GC interleaving in Metronome-TS. Mutators are individually taxed and GC work is concurrent.

is best-suited to periodic applications and is fragile under overload conditions.

In this paper we introduce the *tax-and-spend* scheduling policy, which subsumes previous approaches to RTGC (time-based, work-based, and slack-based). A RTGC using tax-and-spend supports the full range of system and workload requirements detailed above. Tax-and-spend requires a collector architecture that is both *incremental* (breaking collection work into quanta lasting at most a few hundred microseconds) and *concurrent* (allowing collection to occur in parallel with application execution). To exploit multiprocessors efficiently it must also be *parallel* (allowing collection work to be done by multiple threads). We implemented tax-and-spend in a significantly revised version of Metronome called Metronome-TS. Metronome was a good starting point for this work because it is already incremental and somewhat parallel.

2. COLLECTOR DESIGN PREREQUISITES

Tax-and-spend requires its collector to be both incremental and concurrent. A non-concurrent incremental collector (e.g. Metronome) performs garbage collection with all application ("mutator") threads paused for very short intervals, as shown in Figure 1. Tax-and-spend requires the alternative organization shown in Figure 2. Both figures show two mutator threads and two dedicated GC threads. In Metronome all GC work was done on separate threads to facilitate fine-grained scheduling. Metronome-TS makes two essential changes. First, GC work occurs on the GC threads concurrently with mutator execution. Second, when mutators do need to stop, they can be *taxed* by requiring them to perform short quanta of collector work themselves (but with different mutators doing this work at different times, concurrently with the execution of other mutators).

Taxation is needed when the degree of load on the system makes it essential to steal small quanta of time explicitly from mutator threads. Concurrent collection by itself is insufficient, since it relies on the operating system to interleave GC activity with mutator activity, which does not provide enough scheduling precision to meet real-time guarantees and prevent heap exhaustion. Taxation also provides robustness to overload, as we will describe below.

Taxation requires frequent checks in the JVM's code paths for pending memory management work. Metronome-TS imposes taxation in the allocation "slow path" (the path taken by a mutator when it cannot allocate from per-thread data structures and must replenish them from global data structures). This is supplemented when needed by forcing taxation at *safe points*, which are places where mutator stacks and program counters are consistent. To achieve safe point semantics, threads frequently check a flag in the thread's control structure, and if that flag is set, they invoke a safe point callback function. In Metronome, this callback mechanism was used to stop all the threads. In Metronome-TS we use it to ensure that each thread (individually) is taxed fairly and does critical tasks in a timely fashion.

Generally speaking, the tax-and-spend scheduling methodology requires that a collector is at least both concurrent and highly incremental (with maximum collection quanta well below the WCET of the intended workload). For good performance on multiprocessors, even with modest numbers of processors, the capability to perform collection in parallel is also required. Metronome-TS has all three properties: it is concurrent, parallel, and highly incremental (down to a granularity of about 200μs).

Remaining subsections of this section describe some key algorithms that were used to make the already incremental Metronome into the concurrent Metronome-TS. The treatment is not exhaustive (many engineering changes too diverse to cover in detail were also required), nor is it prescriptive (there may be other ways of achieving a collector that is simultaneously incremental, concurrent, and parallel and hence schedulable via the tax-and-spend technique). Aspects of Metronome-TS that are substantially the same as Metronome are omitted in the interest of space. A description of the Metronome system is available in [4], and the basic principles of the Metronome collection algorithm are in [7].

2.1 Global Agreement with Ragged Epochs

In any parallel or concurrent collector there will be a requirement for agreement amongst threads, including mutator threads. In Metronome, agreement was readily achieved (as long as it could be achieved in a time shorter than the GC quantum) since the collector had exclusive control when doing GC work and all mutators were paused at safe points.

Metronome-TS substitutes a *ragged epoch* protocol in places where Metronome relied on exclusive control to achieve agreement. The protocol relies on a single global epoch number that can be (atomically) incremented by any thread, and a per-thread local epoch number. Each thread updates its local epoch by copying the global epoch, but it does so only at safe points. Thus, each thread's local epoch is always less than or equal to the global one. Any thread can examine the local epochs of all the other threads and find the least local epoch value, called the *confirmed epoch*. The confirmed epoch is always less than or equal to the global epoch.

Suppose thread *A* has performed some operation that establishes a new state of the memory management system (for example, it has enabled a store barrier that must be honored by all mutator threads during the next phase of the collection). Thread *A* then increments the global epoch and notes the result. Only when the confirmed epoch equals (or passes) this remembered value can *A* be certain that no thread is computing with a too-early view of the state (e.g., a view in which the store barrier is not enabled). For hardware with weak memory ordering, it is necessary for a thread to insert a memory barrier prior to updating its local epoch.

Implementing this protocol involves one subtlety. Waiting for all threads' local epochs to catch up can be delayed indefinitely when some threads are in I/O waits. However, all such waits occur in native methods. When the JVM detects that a thread is entering a native method, it ensures that the thread is at a safe-point and releases the thread's *VM Access* property. Threads without VM Access do not have access to any JVM data structures and can be

thought of as potentially up to date with the global epoch (they will copy it as soon as they reacquire VM Access and before doing anything to make their view of the memory management system inconsistent).

Thus, the confirmed epoch is calculated using only threads with VM Access. There is then a short window during which a thread may have just regained VM Access but not yet updated its local epoch. These briefly laggard local epochs are rendered harmless by remembering the previous confirmed epoch while calculating the new one and not allowing the confirmed epoch to move backwards.

Threads may also lag due to overload or priority pre-emption. This is handled by the boosting mechanism described in Section 2.4.

2.2 Phase Agreement Using "Last Man Out"

Garbage collection is inevitably divided into phases like marking and sweeping. In the architecture described here, there are more such phases than one might at first think, due to two factors. First, to achieve incrementality, everything must be broken down into small steps. A conceptually unified action (such as effecting the transition from marking to sweeping), gives rise to many distinct actions. Second, the full Java language has constructs, such as finalization, weak and soft references, and string interning, which interact in complex ways with memory management. These constructs add phases to the collection cycle: for example, reference clearing can only be considered once marking is over and it is known which objects are reachable. In Metronome-TS there are, in fact, more than thirty phases in the collection cycle.

We require agreement on when one phase of a collection ends and another begins. Metronome achieved such agreement readily because all GC work occurred on dedicated threads which could block briefly as long as enough time remained in the GC quantum. In Metronome-TS, the agreement mechanism must be strictly non-blocking because some of the work is being done by mutator threads, each of which can be at a different point within its taxation quantum. The ragged epoch mechanism was not efficient for this form of agreement since it did not distinguish threads currently doing GC work from other irrelevant threads. Instead, we employed a *last man out* protocol for phase transition agreement. The protocol works by storing a phase identifier and a worker count in a single atomically updatable location.

When a mutator thread is potentially eligible for taxation, it atomically increments the worker count, leaving the phase identifier the same. When a mutator exhausts the work quantum while there is still work left to do in the phase, it atomically decrements the worker count, also leaving the phase identifier the same. When any thread detects that there is (apparently) no more work left in the phase and also that the current worker count is one, the exiting worker changes the phase atomically with decrementing the worker count to zero. This establishes the new phase.

The last man out protocol is a sound basis for phase transition as long as there is no "invisible work" (that is, there is some global work queue and any incomplete work is returned to that queue when a worker exits). Eventually, there will truly be no more work to do and some thread will end up being the last man out and able to declare the next phase.

The requirement of making all work visible to the last man out in a parallel phase is easily met except in one case. This special case is the termination of the marking phase. The Metronome collectors [7, 4] use Yuasa's snapshot-at-the-beginning technique [27], and hence they rely on a write barrier. For that reason, termination of marking requires coordinated checking of the barrier buffer in conjunction with the normal work queue. This coordinated checking is difficult to accomplish in a distributed fashion across multiple worker threads. Consequently, when marking "appears to be over," as observed by individual threads, there is a transition to a phase in which marking is temporarily handled by one thread, which uses the ragged epoch mechanism to detect whether there is global agreement that all write buffers are empty. The deciding thread can declare a false alarm and return to parallel marking. Eventually termination conditions are met (since snapshot algorithms are inherently bounded and monotonic) and the single deciding thread moves to the first post-marking phase.

2.3 Forcing Per-thread Actions

While most phases of a collection cycle just need enough worker threads to make progress, some require that something be done by (or to) every mutator thread. At the start of collection, every thread's stack must be scanned. At other times, thread-local caches must be flushed to make information visible to the collector. To support this capability, some phases are marked as *callback* phases. As soon as there is agreement that the collection is in a callback phase, a distinct callback protocol takes temporary precedence over last man out.

In a callback phase, a GC master thread frequently examines all mutator threads to see whether they have accomplished the required task. For each thread that has not, a different action is taken depending on whether the thread has VM Access. Threads with VM Access are requested to call back at their next safe point and to perform the required action (stack scanning, cache flushing, etc.) on themselves. Threads that do not have VM Access are briefly prevented from reacquiring VM Access while the callback action is performed on their behalf by a GC thread. Thus, the maximum delay imposed on any thread during a callback phase is the time taken to perform the action, regardless of whether the thread does or does not have VM Access at the time the phase begins. Threads without VM Access are often in I/O waits and hence there is a good chance that the thread will avoid delay entirely.

2.4 Ensuring Progress with Priority Boosting

Progress is a fundamental requirement for a real-time collector, because it must finish collection before the heap is exhausted. All three of the protocols we have used in Metronome-TS (ragged epochs, last man out, and callback) can have their progress impaired when a thread that needs to be heard from has a lower priority than other threads that are monopolizing the available processors. This problem is a case of the well-known priority inversion phenomenon, but common solutions such as priority inheritance are not applicable because garbage collection is a complex global operation lacking a single kernel-managed resource. Instead, we use the kernel's manual priority setting mechanism to boost the priorities of threads that are delaying progress and then revert them to their normal priority as soon as possible thereafter.

Another kind of priority inversion may occur when a mutator thread has to perform collector work while holding a Java lock; we describe this in more detail along with a solution in Section 4.

3. TAX-AND-SPEND SCHEDULING

We now present a new approach to scheduling of real-time garbage collection that is sufficiently flexible and general to handle the variations in both application types and system configurations that we described in the introduction.

Our approach provides a clean policy/mechanism separation and subsumes most of the approaches used previously, including time-, work-, and slack-based schedulers. It also allows a simple default configuration which is robust and performs well for the majority of users.

3.1 Minimum Mutator Utilization

Collector scheduling in Metronome-TS is based on the *Minimum Mutator Utilization* (MMU) metric of Cheng and Blelloch [12]. The MMU of a system at time resolution w, $MMU(w)$, is defined as the minimum fraction of time available to mutator(s) over *any* time window of size w during the entire execution of the system.

Thus if a task with a deadline d and a worst-case execution time e (where $e < d$) runs during garbage collection, it will meet its deadline if $MMU(d) \geq e/d$.

MMU is simple for users to reason about because they can just consider the system as running somewhat slower than the native processor speed until the responsiveness requirements become very close to the quantization limits of the collector.

As a metric, MMU is superior to maximum pause time, since it accounts for clustering of individual pauses, which can cause missed deadlines and pathological slowdowns.

As described below, Metronome-TS uses MMU in a much more general way than collectors like Metronome [7]: different threads may run at different MMUs, background threads on spare processors may be used to obtain very high MMUs on mutator threads, and the units of time can be either physical or virtual.

3.2 Per-thread Scheduling

A key aspect of our system is that measurement and scheduling of collector work is driven using per-thread metrics, and collector quanta may be performed on mutator threads.

This has a number of advantages: first, it allows all collector-related activity to be tracked and accounted for, even things that of necessity occur on mutator threads (such as obtaining a new write buffer, formatting a page, and so on). It may be arguable whether these operations are part of "garbage collection." However, they are memory management operations that perturb mutator execution and therefore the scheduler should track them so that it avoids scheduling too much work on that thread.

Secondly, by performing collector quanta on mutator threads, we avoid perturbations of the scheduler caused by operating system assumptions about how it should behave when a thread voluntarily yields, particularly when the collector is scheduled at a considerably finer quantum size than the operating system (in our case, 200μs rather than 10ms). For example, say we are attempting to interleave one collector and one mutator thread at 200μs granularity. In an unloaded system, this generally works well. But in a loaded system, the operating system will assume that since both threads "voluntarily" (from its point of view) yielded the CPU, they are done with their *operating system* quanta, and it may run some other unrelated thread for its full 10ms. With round-robin scheduling, this will lead to the JVM getting only 4% of the CPU, instead of 50%. By interleaving at a fine granularity on the mutator thread, such effects are avoided and the two schedulers operate in a more orthogonal and compositional manner.

Thirdly, it allows different threads to run at different utilization levels, providing flexibility when allocation rates vary greatly between threads, or when particular threads must be interrupted as little as possible (for instance, on interrupt handlers).

Finally, it allows quantization overhead to be reduced for threads with less stringent timing requirements.

3.3 Tax-Based versus Slack-Based Scheduling

The most fundamental distinction between approaches to scheduling real-time collection is between *tax-based* and *slack-based*. In a slack-based scheduler, exemplified by the real-time collector of Henriksson [15], the collector runs during "slack" periods when the mutator threads have voluntarily yielded the processor. There may also be additional low-priority mutator threads that run when neither collector nor high-priority mutator threads are running.

Slack-based scheduling uses a paradigm familiar to programmers of classical periodic real-time systems and works very well in such an environment. However, it can cause catastrophic failure in overload situations, and it is thus poorly suited to queued, adaptive, or soft interactive real-time systems.

On the other hand, tax-based schedulers periodically interrupt the mutator to perform a certain amount of work on behalf of the collector. Tax-based schedulers include both the *work-based* approach used by Baker [8] and many subsequent incremental and real-time collectors, as well as the *time-based* approach as exemplified by the Metronome [7] collector.

In a work-based system, a certain amount of collector work is performed every time the mutator performs a certain number of memory-related operations (allocation, read barriers, or write barriers). The amount of collector work is chosen to be proportional to the mutator work such that collection will finish before memory is exhausted.

Work-based scheduling is relatively simple to implement and does not rely on real-time operating facilities such as high-precision timers. However, it often suffers from poor MMU and considerable variation in pause times (in one study, Bacon et al. showed that the MMU of a work-based schedule did not exceed 50% until the time window reached half a second [7]). Thus work-based scheduling is unsuitable for any but the "softest" of real-time systems.

Time-based scheduling taxes the mutator thread based on the passage of physical time by directly honoring an MMU requirement ($MMU(w) = u$) and interleaving mutator and collector activity for given amounts of physical time.

Time-based scheduling allows simple but reliable reasoning about the timing behavior of the system. It works well for queued, periodic, and interactive real-time systems and is robust in the presence of overload because taxes continue to be assessed. Reasoning about timely completion is simpler than with a slack-based scheduler, but somewhat more complex than with a work-based collector (although in practice we have found that the vast majority of applications work with an MMU of 70%). However, in a system with slack, it introduces more jitter than a slack-based collector, since collection may occur at any point so long as MMU requirements are obeyed. It also requires high-precision timing facilities.

3.4 Unifying Tax and Slack Scheduling

Since both tax-based and slack-based scheduling have desirable properties, we would like to unify them in a single *mechanism* that combines their advantages and allows the application of various *policies*.

We unify the two mechanisms using an economic analogy: each mutator thread is subject to a *tax rate*, which determines how much collection work it must perform for a given amount of execution. This is expressed as a per-thread MMU. Dedicated background GC threads at low or idle priority can run during slack periods and accumulate *credits* for the work they perform. In our system, credits are deposited in a single centralized *bank account*, but more flexible (and complex) policies could make use of multiple bank accounts.

Each thread has its own tax rate (MMU), and the aggregate tax must be sufficient for the garbage collector to finish garbage collection before the system runs out of memory. While the mechanism allows each thread to have an individual MMU, by default all threads have the same MMU, and so far we have found that this works for the majority of applications. However, our music synthesizer [2] required a different MMU for concurrent threads with a very high allocation rate.

3.4.1 Using Slack: Background Threads

The system has some number of collector background threads, usually not more than the number of processors. In practice, background threads are often run at idle priority or are otherwise configured to only run when a CPU is idle. In this way they naturally run during slack periods in the overall system execution.

Background threads execute collector work in a series of work quanta. For each quantum that the background thread executes, it adds the corresponding amount of credit to the bank account.

In the operating system currently targeted by Metronome-TS (the RHEL5 MRG real-time Linux kernel), it is desirable to assign a low real-time priority to the background threads rather than running them at Linux's `SCHED_OTHER` priority. This causes them to be scheduled with a quantum similar to that used by other threads and allows them to exploit idle processors more efficiently. When run at real-time priority, the background threads periodically sleep for a small amount of time. This has a minimal effect on the progress of the collector but makes it possible for ordinary (`SCHED_OTHER`) processes to make progress even when collection within the JVM might otherwise saturate the machine. This is particularly important since it makes it possible for users to log on and kill runaway real-time processes.

3.4.2 Paying Tax: Mutator Threads

The MMU of each mutator thread specifies its maximum tax rate and the granularity at which the rate is applied. Note that the MMU does two things. On the one hand, it ensures that the application can meet its real-time requirements by guaranteeing that the mutator can make sufficient progress in a time window. On the other hand, it allows for the collector to make sufficient progress so that it can complete on time.

The most straightforward approach is for the thread to pay its own tax as it goes, as in a classic tax-based system: it alternately performs mutator work and collector quanta. However, under the tax-and-spend discipline, when a mutator thread is running and the time comes to assess garbage collection tax, it first attempts to withdraw a quantum's worth of credit from the bank. If this operation succeeds, the collector is "ahead" and the mutator can skip its collection quantum and return immediately to the mutator. Thus the mutator only pays tax when there is insufficient collection being performed during slack periods. More generally, if only a partial quantum's worth of credit is available, it can perform a smaller collection quantum and return to the mutator more quickly. The result is that, in the presence of slack, the mutator runs with both higher throughput and lower latencies but without the danger of the collector falling behind. Furthermore, it can also take advantage of *partial slack*.

Tax-and-spend also allows us to treat slack in a uniprocessor and excess capacity in a multiprocessor in a uniform way. If there are four processors and three mutator threads, then a collector thread can run on the fourth processor and bank enough credits that the three threads never have to perform work on behalf of the collector.

But if the tax rate does not divide evenly into the number of processors (for example, a 70% MMU on four processors would require $1\frac{1}{5}$ collector processors), a small amount of collection work will be imposed on the threads running on the other three processors. However, they will still be taxed in such a way that their MMU requirements are met.

3.5 Unifying Time and Work Scheduling

So far we have described the taxation in terms of MMU over time, thus implementing time-based scheduling. However, we can unify both time- and work-based scheduling simply by treating work as a different way of measuring time. Thus time is virtualized and can be measured in absolute physical time, per-thread CPU time, bytes allocated, safe points executed, and so on. Different choices of virtual time axes yield schedules with different tradeoffs in terms of timing precision, overhead, and stringency of requirements on the hardware and operating system.

In fact, even in time-based mode, Metronome-TS uses per-thread CPU time rather than absolute physical time. There are two reasons for this. First of all, since we are performing per-thread scheduling, global time will not work reliably in overload situations and may make the feasibility analysis complex if portions of the system are delayed for non-deterministic amounts of time in blocking system calls (such as I/O waits). Second, and more importantly, it allows users to reason more compositionally about the scheduling behavior of the operating system versus the scheduling behavior of the Java virtual machine.

Since both the standard Linux (RHEL5) and real-time Linux (RHEL5 MRG) environments in which we have implemented the system provide high-precision per-thread timing, we have not implemented some of these more "virtualized" time axes, since they provide far worse timing behavior and are primarily of use in environments like the Windows operating system, which provides no real-time guarantees.

3.6 Oversampling

The original Metronome collector, when given an MMU requirement of $MMU(w) = u$, would schedule one collector quantum of size $w \cdot (1-u)$ every w time units. However, this has three disadvantages. First, any small amount of jitter or overshoot in the scheduler quantum will cause an immediate MMU failure. Second, if the system is capable of using smaller quanta, then users will not receive the improved average-case latency at shorter time scales that may have significant benefits for applications like queued real-time systems. And third, it requires more careful consideration of MMU requirements, complicating the interface for naïve users.

Thus Metronome-TS, like later versions of Metronome, makes use of *oversampling* [6], a technique from audio processing, in which higher fidelity is achieved by using a higher frequency for digital processing than the final output frequency requirement.

Oversampling provides both more accurate MMU at the chosen time window as well as higher (albeit not as tightly guaranteed) MMU at shorter time windows, and it allows much more broadly applicable default settings for the collector.

4. LOCK DEFERRAL

The combination of oversampling and per-thread MMU based scheduling gives Metronome-TS the opportunity to dynamically adjust how GC work is scheduled to be sensitive to application critical sections. The basic scheduling algorithm can be extended as described below to reduce the worst-case execution time of critical sections in certain types of real-time programs by briefly deferring GC work. This is similar in spirit to work on avoiding preemption when operating system locks are held [17].

For this extension to be applicable, the work of the application must be composed of critical and non-critical sections. It works well when the critical sections are short (shorter or similar in length to a GC quantum) and at the time scale of a GC scheduling window there must be sufficient flexibility (due either to idle time or non-critical section work) to make it feasible to schedule the required GC work outside of the critical sections.

The key observation is that oversampling divides an MMU time window into a number of smaller units at which GC operations are scheduled. For example, a 4ms MMU window with 200us GC

Benchmark	GCs	MMU% @4ms	Physical Pause Times (μs) Max	Avg	StDev	Memory Load (%)	Memory Work (%) Alloc	Mutator	Background	Boosted (%)	Ragged (per GC)
DaCapo (no background GC threads)											
antlr	9	68.17	359.22	15.12	47.08	10.49	29.86	63.1	7.04	0.09	42.1
bloat	43	67.85	363.29	18.96	52.58	18.11	26.58	68.08	5.34	0.1	50.2
chart	40	66.7	359.46	26.61	62.23	9.45	21.54	67.8	10.66	0.0	110
eclipse	163	67.37	328.41	30.51	66.98	21.12	11.99	81.18	6.84	0.12	64.6
fop	3	68.91	390.68	23.53	62.46	8.55	20.26	71.02	8.72	0.03	100.7
jython	39	66.51	364.06	22.86	57.51	17.87	22.89	69.79	7.32	0.17	82.3
luindex	19	68.59	333.82	12.32	41.31	9.75	34.87	58.74	6.39	0.05	44.3
pmd	24	67.37	377.53	33.91	73.33	19.67	16.65	77.48	5.87	0.04	147.7
hsqldb	3	67.41	382.46	41.84	73.29	9.3	18.63	77.28	4.1	0.19	282.7
lusearch	55	60.78	9623.63	139.99	442.29	58.24	2.87	96.69	0.44	0.91	8.7
xalan	32	66.95	7297.89	29.19	83.86	13.41	14.38	82.83	2.79	0.59	12.1
SPECjvm98 and SPECjbb2000 (no background GC threads)											
compress	8	70.09	337.94	26.47	71.01	1.61	7.82	60.74	31.45	0.05	51.8
db	10	68.13	310.05	29.48	70.88	5.06	14.68	69.66	15.66	0.07	210.3
jack	7	68.75	344.67	8.8	31.88	7.83	48.07	48.4	3.53	0.03	40.6
javac	13	68.07	341.15	26.26	66.71	19.7	17.17	77.95	4.88	0.04	65.5
jess	16	67.91	354.47	11.4	39.25	9.58	38.99	57.24	3.77	0.03	49.9
mpegaudio	1	89.95	350.55	29.77	78.29	0.21	2.96	59.71	37.33	0.002901	42
mtrt	9	67.23	369.49	25.91	58.87	22.1	28.32	67.9	3.78	0.07	89.2
jbb	133	72.03	381.18	29.28	63.35	13.29	17.03	80.29	2.68	0.0	109.3
DaCapo (2 background GC threads)											
antlr	9	76.99	128.51	8.12	6.1	14.7	17.21	0.3	82.5	0.16	2.2
bloat	44	83.14	232	8	6.59	24.48	16.25	0.44	83.3	0.13	4.9
chart	43	76.09	231.71	8.92	6.34	12.6	11.84	0.18	87.98	0.0	7.8
eclipse	196	68.61	207.29	7.06	5.89	30.73	7.67	0.2	92.13	0.13	7
fop	2	81.36	103.56	8.34	6.19	10.98	15.05	0.11	84.84	0.08	7.5
jython	56	67.91	279.16	9.59	7.3	28.97	13.54	0.25	86.21	0.2	8.3
luindex	19	81.89	131.2	8.13	5.47	12.78	18.73	0.2	81.07	0.07	3.6
pmd	59	74.24	245.38	9.12	7.74	41.95	5.51	0.31	94.19	0.17	9.7
hsqldb	7	80.76	868.39	21.59	48.99	27.66	2.38	4.07	93.55	0.01343	83.9
lusearch	48	57.23	10268.38	127.98	419.83	60.58	3.34	80.75	15.91	0.88	9.5
xalan	36	67.78	1861	19.66	56.53	25.32	7.27	16.16	76.57	0.42	9.9
SPECjvm98 and SPECjbb2000 (2 background GC threads)											
compress	7	86.39	93.86	9.81	11.91	1.48	1.41	0.43	98.16	0.03	2.3
db	11	87.99	256.98	9.71	6.57	5.95	7.97	0.24	91.79	0.03	8.7
jack	7	87.03	138.21	9.18	5.31	11.83	22.84	0.17	76.99	0.03	6
javac	11	75.61	209.55	7.69	6.02	25.24	13.17	0.26	86.57	0.05	8.6
jess	16	85.34	198	8.67	5.95	13.92	21.28	0.25	78.47	0.05	6.1
mpegaudio	1	89.37	98.67	8.39	12.28	0.43	3.61	1.5	94.89	0.02	3
mtrt	11	87.07	128	10.32	7.17	34.07	16.48	0.19	83.33	0.17	11
jbb	154	80.39	272.14	10.62	7.52	17.25	13.83	0.47	85.7	0.0	35

Table 1: Summary statistics for Metronome-TS on DaCapo, SPECjvm98 and SPECjbb2000 benchmarks

quanta can be divided into 20 scheduling units. With a 70% MMU target, the system should schedule GC work in 6 of these units and allow the thread to execute unperturbed in the other 14. As long as the MMU-derived scheduling constraints are met (both in this window and in the nearby windows), there is flexibility in scheduling the GC work. In particular, if the thread is in a short critical section when the system decides to schedule GC work on the thread, the work could instead be deferred until immediately after the critical section without significantly changing the thread's MMU or the GC's forward progress.

The key implementation issue is to dynamically balance the competing goals of avoiding GC work in critical sections, thus improving the predictability of the application, and scheduling sufficient GC work on the thread, thus ensuring that the GC completes in a timely fashion. We again do this by applying a banking model. Each thread has a private critical section deferral account that is initialized to a GC quantum worth of credits (using the example values above: 200). If the thread is in a critical section when it is asked to perform GC work and it has a positive credit balance, then it is marked as having deferred GC work and is allowed to resume normal execution. When it exits the critical section, its balance is debited by the time the GC work was deferred and it immediately fulfills its GC obligation. If the critical section is long, the thread is asked to perform GC work again before it exits the critical section. If, after deducting the time since it initially deferred the work, its balance is no longer positive then the deferral is canceled. Each time the thread performs GC work, it receives a small credit to its critical section balance. In our implementation we bound the allowable values of the critical section balance to be within 1 GC quantum of zero and give credits at a rate such that an entire GC quantum of credit can be earned in a scheduling window. Using the numbers above, the balance is bounded between -200 and 200 and each time the thread does a quantum of GC work it gets a credit of 33 (200/6).

For the evaluation in section 5.4, we used the simple heuristic that critical sections are indicated by the thread holding a Java-level monitor. The JVM's monitorenter/exit sequences were augmented to increment/decrement a per-thread nesting count and to call into the GC when a thread that was marked as having deferred GC work exits its outermost locked region.

5. EVALUATION

We have implemented Metronome-TS in the IBM production J9 Java virtual machine. The implementation began with a development version of the implementation of the Metronome collector in the IBM WebSphere Real Time product [4], which is a parallel (but

non-concurrent) version of the original [7] Metronome algorithm. This code base fully supports all Java 5 language features including finalization, class unloading, and weak, soft, and phantom references. However, for our experiments we have disabled RTSJ (JSR-1) support in the JVM and therefore the Metronome performance results are not comparable to the WebSphere Real Time product.

Both Metronome and Metronome-TS are implemented in the exact same base JVM, and share much of the common GC code and data structures. Thus, comparisons between the two algorithms are made on a level playing field with the only differences in the systems being the GC algorithms themselves.

For Metronome-TS, we use per-thread CPU consumption as reported by the operating system as the virtual time axis to compute its per-thread MMU. Metronome uses physical time and a global MMU.

For all measurements we set a utilization target of MMU(4ms)=70% and use a nominal collector quantum of 400μs for Metronome and 200μs for Metronome-TS. The flexibility of the tax-and-spend scheduling module enables smaller quanta for Metronome-TS than are feasible with the more rigid scheduling of Metronome.

5.1 Experimental Environment

Experiments were performed on an 8-processor IBM LS41 Blade containing four dual-core 2.4 GHz AMD Opteron 8216 processors, each of which has a 1MB unified L2 cache. The total system memory was 16 GB. For experimental purposes, one processor was dedicated to collection of trace data to minimize interference from measurement. We also reserved 4 GB of memory as a RAM disk (/dev/shm) to avoid perturbation by I/O due to trace collection. Thus we effectively used a 7-processor machine with 12 GB of memory for the evaluations.

The operating system was Red Hat Enterprise Linux 5 Real Time edition (RHEL5 MRG). RHEL5 MRG is a 2.6.24 Linux SMP kernel with the PREEMPT RT patch set, running in 64-bit mode.

Evaluations were performed using a combination of well-known Java benchmarks: DaCapo, SPECjvm98, and SPECjbb2000[1]. To factor out interference and variation due to the JIT compilation, each run involved multiple iterations of the benchmark. During the warmup iterations, JIT compilation happened normally. The harness then invoked `Compiler.disable()` to turn off the JIT compiler, thus ensuring that in the measurement period no further JIT compilation occurred.

In addition, we evaluated a complete application: the control program for the JAviator quad-rotor helicopter [3] (10 KLOC). For this evaluation, the control program did not use the Exotasks system but was reorganized to depend only on real-time garbage collection for its low-latency scheduling requirements. The application was run on the same hardware as the other benchmarks, using a simulated helicopter running on a separate processor.

5.1.1 Data Collection

To fully understand and evaluate the performance of the complete system, we gathered traces from the operating system, JVM, and application with the TuningFork [5, 26] system. TuningFork provides trace collection facilities which we used to instrument the JVM and the applications. It also includes a facility for gathering Linux traces via the System Tap [22] facility.

The TuningFork infrastructure allows us to collect highly detailed yet compact traces that show all collector-related activity in the JVM and all scheduling decisions by the operating system. We use this to compute statistics that are based on the physical times when processes are running, not counting such things as operating system interrupt handlers.

[1]The reported results do not directly or indirectly represent a SPEC metric as the benchmarks were run with modified harnesses to enable the detailed collection of GC and application metrics.

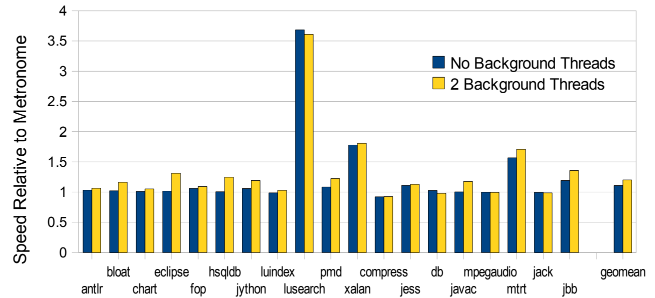


Figure 3: Speedup over Metronome of Metronome-TS with and without background threads.

5.2 Collector Performance

To begin with, we examine basic metrics for real-time collection across the standard benchmarks from the DaCapo and SPECjvm98 suites, as shown in Table 1. We evaluate mutator utilization, pause times, distribution of memory system work (both allocation and collection), and the effects of priority boosting.

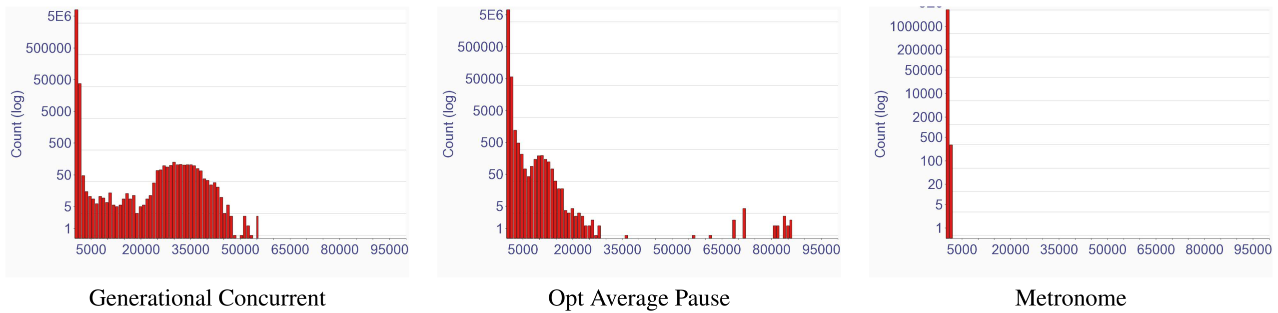
For each set of benchmarks, we show results with no background GC threads and with two background GC threads. Without background threads, the mutators are being taxed to perform all collection work, regardless of whether spare CPU capacity exists. With background threads, collection work should move to the spare processors, except in the presence of overload. The contrast sheds light on how the system performs when CPU resources either do or do not exceed the multiprogramming level.

The system maintains per-thread mutator utilizations that include all work done in support of the memory system by the mutator. This includes collector quanta when they are being performed by the mutator (as when there are no background threads) as well as stack snapshots (when collection begins), write barrier buffer management, and asynchronous operations for the ragged epochs. We also charge all allocation-related activity (other than simple list dequeue operations) against the MMU of the mutator. Previous work on garbage collection has tended to ignore the MMU effect on mutators caused by allocation and auxiliary work; thus our MMU results are significantly more stringent than those of previous work.

Three of the DaCapo benchmarks (`hsqldb`, `lusearch`, `xalan`) have more than 7 concurrent threads, and they therefore ran in overload conditions. The `lusearch` benchmark has 32 simultaneous threads at the same priority and allocates at an extremely high rate (about 150 MB/s). Thus these three benchmarks can not really achieve real-time behavior, and we will treat them separately.

In the absence of overload, all programs achieve an MMU of at least 66% at a 4ms time window. For comparison, Metronome requires a 10ms time window to achieve a comparable MMU. Only `lusearch` misses its target significantly, with an MMU of 57%.

The next three columns in Table 1 show the duration of pause times in physical time: that is, wall clock time from the beginning of a collector quantum until its end. In the absence of overload this is the maximum time the application is interrupted by the collector. Without background threads the maximum pause is 391μs; with background threads the mutator threads do far less collector-related work, and the maximum pause drops to 279μs. Note that with background threads the average pause time and the standard deviation also drop considerably.



Generational Concurrent

Opt Average Pause

Metronome

Table 2: SPECjbb2000 transactions time distributions for three different J9 garbage collectors. Note that the y-axis is logarithmic and the x-axis range on each graph is from 0 to 100 milliseconds..

In the presence of overload, physical time pauses are as much as 10.3ms. However, this is simply due to the fact that the operating system deschedules the process in the midst of a collector quantum in order to allow other eligible threads at the same priority a share of the CPUs.

The next column gives the *memory load*—that is, the percentage of total CPU time spent in either allocation or garbage collection. The following 3 columns show the component parts of the memory load: mutator allocation, mutator collection, and background collection. The memory load varies greatly across applications. However, with background threads the mutator threads always spend less than 1.5% of their time performing collector work, although they may still spend significant amounts of time on allocation. On the other hand, the total memory load increases with background threads because the collector runs more frequently, using the spare cycles to keep memory utilization low.

The last two columns of Table 1 relate to the need for priority boosting to ensure that lack of global progress by the collector does not create a meta-level priority inversion. In the absence of overload, mutator threads never spend more than 0.2% of their time at a boosted priority, since they generally reach "safe points" quickly and allow the collector to make progress through its phases, as controlled by the ragged epochs. However, in the presence of overload, threads spend as much as 0.91% of their time boosted, since the collector must occasionally force threads to "roll forward". Nevertheless, this is still a modest fraction of the total time.

Figure 3 shows the throughput performance of Metronome-TS relative to that of Metronome. Overall, not only are the applications achieving significantly better real-time behavior (and with a more stringent definition of GC work in the MMU computation), but throughput is improved as well. The mean speedup is 11% with no background threads and 20% when background threads are used to exploit excess CPU capacity by offloading GC work. The only slowdowns are 8% on `compress` and 1% on `db` with background threads. These two benchmarks are the only ones in our suite where a significant fraction of the GC work is scheduled via safe points instead of allocation slow paths. We speculate that the slowdowns could be reduced by fine tuning this mechanism.

5.3 SPECjbb2000 Case Study

This section presents an in-depth analysis of the SPECjbb2000 benchmark to illustrate how Metronome-TS performs on a simplified transaction processing workload. To gather the data, we added TuningFork instrumentation to jbb to time individual transactions.

Figure 2 and Table 3 show the distribution of individual transaction times for a 4 minute, 4 warehouse run of jbb using three different J9 garbage collectors. The two non-realtime collectors

Garbage Collector	Transaction Times (microseconds)				
	Median	99.9%	99.99%	99.999%	Max
GenCon	39	1,081	31,654	41,550	55,462
OptAvgPause	38	1,491	9,781	15,434	86,093
Metronome	114	893	997	1,098	1,756
Metronome-TS	113	601	649	694	762
Metronome-TS(2 bg)	109	384	437	565	733

Table 3: SPECjbb2000 Transaction Times

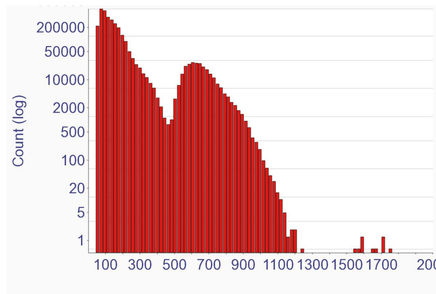
are generational and non-generational concurrent mark/sweep collectors (IBM SDK J2SE Java 5 SR 5). They have excellent average case performance, but suffer from very large outliers. Note that the y-axis is logarithmic to make the outliers observable. In contrast, average case performance on Metronome is 3x lower, but worse case behavior is 1-2 orders of magnitude better than the other collectors.

Figure 4 compares Metronome (top graph) and various configurations of Metronome-TS. The Metronome data is identical to the bottom graph of Figure 2, but shown at a 50x finer scale. The "humps" in the Metronome graph are caused by transactions that overlap with its non-concurrent 400μs GC quantum. The second and third graph show Metronome-TS with zero and two background threads respectively. Even without background threads, Metronome-TS does significantly better than Metronome. Both the shorter quantum (200 μs) and the more precise tracking of MMU to include allocation and write barrier slow path operations as GC work contribute to the improvement. Since there is spare capacity in this scenario, Metronome-TS is able to off-load virtually all GC work to the background threads and achieve the smooth distribution shown in the third graph. Metronome-TS is able to obtain significantly better real-time behavior than Metronome and also significant throughput improvements: 19% better without background threads and 36% with them.

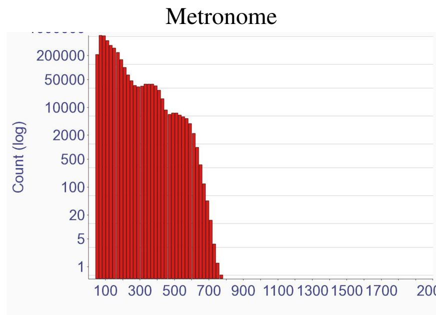
The fourth and fifth graphs of Figure 4 show transaction times when the system is loaded with additional CPU-bounded processes running at normal priority and at a real-time priority between that of the background threads and the jbb warehouse threads. The key observation is that, under normal load, the background threads continue to function (as shown by the similarity of third and fourth graphs), but under real-time load the background threads smoothly get out of the way (shown by the similarity of the second and fifth graphs).

5.4 Effectiveness of Lock Deferral

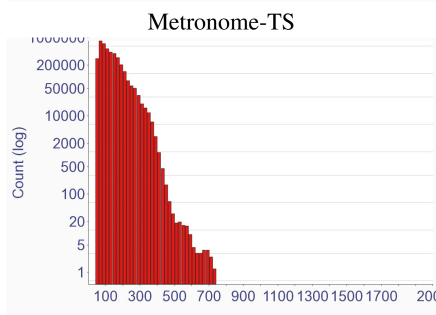
To isolate the potential benefits of deferring GC work units when application threads are in short critical sections, we wrote a sim-



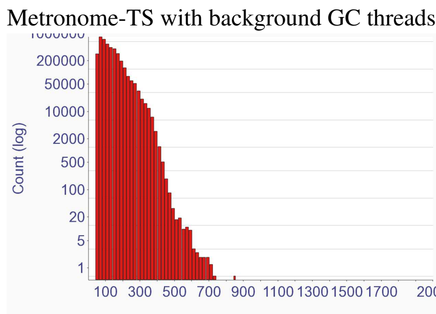
Metronome



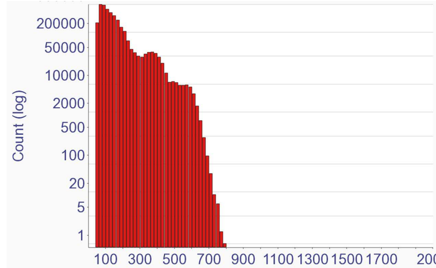
Metronome-TS



Metronome-TS with background GC threads



Metronome-TS with background threads and **SCHED_OTHER** load



Metronome-TS with background threads and real-time load

Table 4: SPECjbb2000 transactions time distributions with real-time collectors. The y-axis is logarithmic while the x-axis ranges from 0 to 2000 microseconds. The Metronome graph shows the same data as Figure 2, but at a 50-times finer scale.

	Critical Section Length (microseconds)				
	Median	Std Dev	99%	99.9%	Max
Deferral Enabled	224	15.1	271	290	363
Deferral Disabled	221	56.1	735	807	854

Table 5: Impact of GC Work Deferral in Critical Sections.

ple synthetic workload in which worker threads spend 5% of their time executing in critical sections. They perform the same basic computation in both critical and non-critical sections: allocating memory at a rate of 1.8MB/sec, mutating the heap, and otherwise computing. The critical sections are sized to be the same order of magnitude as the GC quantum (250 microseconds).

Table 5 reports the impact of GC work deferral on critical section length. The program was run for 5 minutes with and without deferral. In both runs, over 63,000 critical sections were completed and 190 GC cycles occurred. Overall throughput, median critical section length, MMU, and peak and average memory usage were virtually identical between the two runs. The only significant difference can be seen in the greatly reduced standard deviation, 99%, 99.99%, and max values of the critical section lengths acheived when deferral was enabled. These improvements in predictability are enabled by relatively small variations in the scheduling of GC work: work was only actually deferred out of 1.2% of the executed critical sections.

5.5 The JAviator Control Application

We ran the JAviator control application on both Metronome and Metronome-TS, measuring the times between successive readings of the gyro sensor data. For optimal performance of the control, these readings needed to occur 20ms apart. There is a certain amount of natural jitter in this application due to the behavior of the simulated sensor and the fact that the simulation was running on a separate machine (the same jitter occurs in the real JAviator due to communication delays from the actual gyro). Thus, the differences in the mean and standard deviation between the two runs were not dramatic: mean=20.028, std.dev=0.368 for Metronome-TS, as opposed to mean=20.059, std.dev=0.448 for Metronome. However, the maximum was only 20.059ms for Metronome-TS as opposed to 23.739ms for Metronome, and the several larger outliers in the latter coincided with garbage collection quanta. Thus, we have direct evidence that the difference between these algorithms can have a direct impact on a real application where timing differences can directly affect the stability of a device under control.

6. RELATED WORK

Real-time garbage collection was first explored by Baker [8], followed by numerous others [19, 14, 1, 24, 27, 15]. The minimum mutator utilization metric was introduced by Cheng [11] and exploited by Metronome [7], which also had a strict bound on memory consumption. Although the implementation of Metronome-TS evaluated in this paper did not include compaction, suitable approaches have been proposed [18, 20]. A commercial implementation of the Metronome algorithm is part of the IBM WebSphere Real Time VM [4]. Azul [13] and BEA [9] are production systems that also provide significantly reduced pause times. Ovm [21] is a research prototype that includes a collector based on Metronome but does not implement the complete Java specification.

The tax-and-spend approach that Metronome-TS uses to schedule collector work is an example of an economic model. Such models are becoming common in distributed systems (e.g. [10]) where the actors in the model economy have analogs in the real money

economy. They have also been used in a less externalized way (e.g. in databases [25]) as a way of managing complex cost/benefit tradeoffs. Here, we have specialized this technique for the problem of integrating slack-based and tax-based scheduling in a garbage collector.

Metronome-TS uses priority boosting to ensure progress when there is significant competition for processor resources, overcoming a form of *priority inversion*. This phenomenon was first recognized as a problematic behavior of monitors in the presence of priority scheduling [16]. The technique of *priority inheritance* [23] was developed as one solution. Priority boosting achieves the effects of priority inheritance manually, which is needed in the case of Metronome-TS since it lacks a concrete single resource required to activate kernel-assisted mechanisms.

7. REFERENCES

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

[2] AUERBACH, J., BACON, D. F., BÖMERS, F., AND CHENG, P. Real-time music synthesis in Java using the Metronome garbage collector. In *Proc. International Computer Music Conference* (Copenhagen, Demmark, 2007).

[3] AUERBACH, J., BACON, D. F., IERCAN, D. T., KIRSCH, C. M., RAJAN, V., RÖCK, H., AND TRUMMER, R. Java takes flight: Time-portable real-time programming with Exotasks. In *Proc. ACM Conference on Languages, Compilers, and Tools for Embedded Systems* (2007). *SIGPLAN Notices*, *42*, 7, 51–62.

[4] AUERBACH, J., ET AL. Design and implementation of a comprehensive real-time Java virtual machine. In *Proc. International Conference on Embedded Software* (Salzburg, Austria, 2007), pp. 249–258.

[5] BACON, D. F., CHENG, P., FRAMPTON, D., GROVE, D., HAUSWIRTH, M., AND RAJAN, V. On-line visualization and analysis of real-time systems with TuningFork (demonstration). In *Compiler Construction* (Vienna, Austria, Mar. 2006), vol. 3923 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 96–100.

[6] BACON, D. F., CHENG, P., GROVE, D., AND VECHEV, M. T. Syncopation: generational real-time garbage collection in the metronome. In *Proc. ACM Conference on Languages, Compilers, and Tools for Embedded Systems* (New York, NY, USA, 2005), ACM, pp. 183–192.

[7] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proc. Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

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

[9] BEA. BEA WebLogic real time: Predictable mission-critical performance for java - today. Technical white paper available at http://www.bea.com, Jan. 2006.

[10] BUYYA, R., ABRAMSON, D., GIDDY, J., AND STOCKINGER, H. Economic models for resource management and scheduling in grid computing, 2002.

[11] CHENG, P. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie-Mellon Univ., Sept. 2001.

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

[13] CLICK, C., TENE, G., AND WOLF, M. The pauseless gc algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (2005), pp. 46–56.

[14] DOLIGEZ, D., AND GONTHIER, G. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conf. Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (Jan. 1994), pp. 70–83.

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

[16] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in mesa. *Commun. ACM 23*, 2 (1980), 105–117.

[17] MARSH, B. D., SCOTT, M. L., LEBLANC, T. J., AND MARKATOS, E. P. First-class user-level threads. *SIGOPS Oper. Syst. Rev. 25*, 5 (1991), 110–121.

[18] MCCLOSKEY, B., BACON, D. F., CHENG, P., AND GROVE, D. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. Tech. rep., IBM Research, Apr. 2008.

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

[20] PIZLO, F., PETRANK, E., AND STEENSGAARD, B. A study of concurrent real-time garbage collectors. *SIGPLAN Not. 43*, 6 (2008), 33–44.

[21] PIZLO, F., AND VITEK, J. An emprical evaluation of memory management alternatives for real-time Java. In *Proc. 27th IEEE International Real-Time Systems Symposium* (2006), pp. 35–46.

[22] PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. Locating system problems using dynamic instrumentation. In *OSL '05: Proceedings of the Linux Symposium* (2005), vol. 2, pp. 49–64.

[23] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39*, 9 (1990), 1175–1185.

[24] SIEBERT, F. Eliminating external fragmentation in a non-moving garbage collector for Java. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (San Jose, California, Nov. 2000), pp. 9–17.

[25] STONEBRAKER, M., DEVINE, R., KORNACKER, M., LITWIN, W., PFEFFER, A., SAH, A., AND STAELIN, C. An economic paradigm for query processing and data migration in mariposa. In *Proc. International Conference on Parallel and Distributed Information Systems* (1994), pp. 58–68.

[26] TuningFork Visualization Platform. *tuningforkvp.sourceforge.net*.

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