High-level Real-time Programming in Java

David F. Bacon Perry Cheng David Grove Michael Hind V.T. Rajan Eran Yahav
IBM T.J. Watson Research Center

Matthias Hauswirth
Università della Svizzera

Christoph M. Kirsch
Universität Salzburg

Daniel Spoonhower
Carnegie Mellon University

Martin T. Vechev
University of Cambridge

ABSTRACT

Real-time systems have reached a level of complexity beyond the scaling capability of the low-level or restricted languages traditionally used for real-time programming.

While Metronome garbage collection has made it practical to use Java to implement real-time systems, many challenges remain for the construction of complex real-time systems, some specific to the use of Java and others simply due to the change in scale of such systems.

The goal of our research is the creation of a comprehensive Java-based programming environment and methodology for the creation of complex real-time systems. Our goals include construction of a provably correct real-time garbage collector capable of providing worst case latencies of 100 μs, capable of scaling from sensor nodes up to large multiprocessors; specialized programming constructs that retain the safety and simplicity of Java, and yet provide sub-microsecond latencies; the extension of Java's "write once, run anywhere" principle from functional correctness to timing behavior; online analysis and visualization that aids in the understanding of complex behaviors; and a principled probabilistic analysis methodology for bounding the behavior of the resulting systems.

While much remains to be done, this paper describes the progress we have made towards these goals.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems; D.3.2 [Programming Languages]: Java; D.3.3 [Programming Languages]: Language Constructs and Features—Dynamic storage management; D.3.4 [Programming Languages]: Processors—Memory management (garbage collection) D.4.7 [Operating Systems]: Organization and Design—Real-time systems and embedded systems

General Terms: Experimentation, Languages, Measurement, Performance

Keywords: Scheduling, Allocation, WCET, Tasks, Visualization

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

1. INTRODUCTION

Real-time systems are rapidly becoming both more complex and more pervasive.

Traditional real-time programming methodologies have revolved around relatively simple systems. This has meant that it was possible to use restrictive programming methodologies with deterministic, statically verifiable properties or very low-level programming techniques amenable to cycle-accurate timing analysis [18].

However, those techniques do not scale to the large, complex systems that are now beginning to be built. The lack of scaling is manifested at both the theoretical and the practical level. Basic principles of undecidability mean that as the software increases in size, the required expressiveness yields a system that can not be statically verified. Basic principles of software engineering mean that low-level programming is untenable at the resulting scale.

As a result there is broad interest in using Java for a wide variety of both soft- and hard-real-time systems. Java provides two main advantages: a scalable, safe, high-level programming model, and a huge body of software components that can be used to compose the soft- and non-real-time portions of the system. Being able to use a single language across all domains of the system provides enormous benefits in simplicity, re-usability, and staffing and training requirements.

Java's high level of safety and security comes from a combination of both static and dynamic checking. Although dynamic checks typically reduce execution-time determinism, in a large-scale mission- or safety-critical system the reliability they provide is essential.

The goal of the Metronome project at IBM Research is to make Java suitable for programming real-time systems. Our goal encompasses both hard- and soft-real-time systems, at time scales as low as those achievable by any software system.

The largest potential source of non-determinism in Java is garbage collection. In previous work we addressed this issue with the development of a true real-time garbage collector which is capable of providing latency, utilization, and throughput guarantees that make it suitable for programming systems with periods as low as 5 milliseconds [2].

This technology forms the basis of a new real-time Java virtual machine product being developed by IBM, and is under evaluation by various customers in the defense, telecommunications, and embedded systems businesses.

However, significant challenges remain in real-time garbage collection, in real-time Java, and in complex real-time sys-

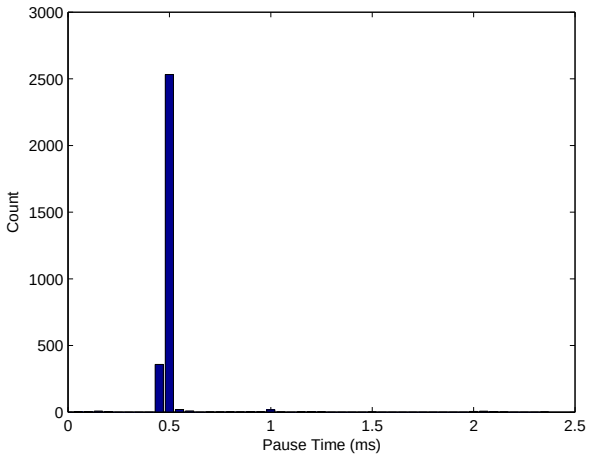


Figure 1: Pause time distributions for javac in the J9 implementation of Metronome, with target maximum pause time of 3 ms and a "beat" size of 500 μs. The actual maximum pause is 2.4 ms.

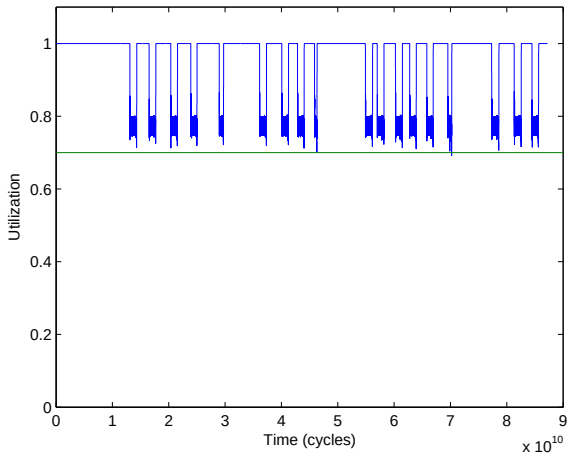


Figure 2: CPU utilization for javac under the Metronome. Mutator interval is 7 ms, collector interval is 3 ms, for an overall utilization target of 70%; the collector achieves this within 0.8% jitter.

tems in general. This paper describes our ongoing research across these various problem domains.

2. THE METRONOME COLLECTOR

In this section we describe our previous work on the Metronome, which forms the heart of the real-time Java system we are developing [2, 1]. Metronome was originally implemented in Jikes RVM [16]; a second generation implementation of the Metronome is underway in IBM's J9 virtual machine. We describe the algorithm and engineering of the collector in sufficient detail to serve as a basis for understanding the work described in the rest of this paper.

The Metronome is an incremental collector targeted at embedded systems. It uses a hybrid approach of non-copying mark-sweep (in the common case) and copying collection (when fragmentation occurs).

The original algorithm was designed for uniprocessors. Recently this restriction has been removed and the system has been applied in server-class systems, as described in Section 3.1.

Metronome uses a snapshot-at-the-beginning algorithm that allocates objects black (marked). Although it has been argued that such a collector can increase floating garbage, the worst-case performance is no different from other approaches and the termination condition is easier to enforce. Other real-time collectors have used a similar approach.

Figures 1 and 2 show the real-time performance of our collector. Pause times are centered around the "beat" which is the nominal frequency of the underlying scheduler ($500\mu s$); worst-case latencies are well below the target. Utilization is high (above the 70%) with minimal jitter. Utilizations of 100% occur while collection is off. In this section we explain how the Metronome achieves these goals.

2.1 Features of the Metronome Collector

Our collector is based on the following principles:

Segregated Free Lists. Allocation is performed using segregated free lists. Memory is divided into fixed-sized pages, and each page is divided into blocks of a particular size. Objects are allocated from the smallest size class that can contain the object.

Mostly Non-copying. Since fragmentation is rare, objects are usually not moved.

Defragmentation. If a page becomes fragmented due to garbage collection, its objects are moved to another (mostly full) page.

Read Barrier. Relocation of objects is achieved by using a forwarding pointer located in the header of each object [8]. A read barrier maintains a to-space invariant (mutators always see objects in the to-space).

Incremental Mark-Sweep. Collection is a standard incremental mark-sweep similar to Yuasa's snapshot-at-the-beginning algorithm [27] implemented with a weak tricolor invariant. We extend traversal during marking so that it redirects any pointers pointing at from-space so they point at to-space. Therefore, at the end of a marking phase, the relocated objects of the previous collection can be freed.

Arraylets. Large arrays are broken into fixed-size pieces (which we call arraylets) to bound the work of scanning or copying an array and to bound external fragmentation caused by large objects.

Since our collector is not concurrent, we explicitly control the interleaving of the mutator and the collector. We use the term *collection* to refer to a complete mark/sweep/ defragment cycle and the term *collector quantum* to refer to a scheduler quantum in which the collector runs.

2.2 Read Barrier

We use a Brooks-style read barrier [8]: each object contains a forwarding pointer that normally points to itself, but when the object has been moved, points to the moved object.

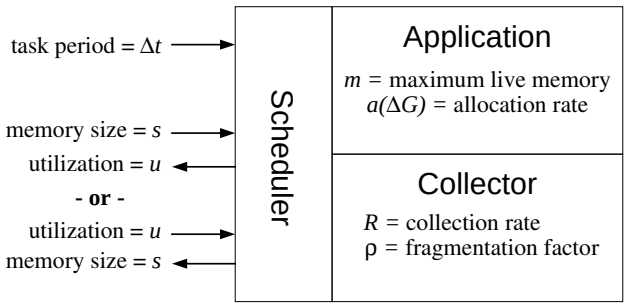


Figure 3: Interaction of components in a Metronomic virtual machine. Parameters of the application and collector are intrinsic; parameters to the scheduler are user-selected, and are mutually determinant.

Our collector thus maintains a to-space invariant: the mutator always sees the new version of an object. However, the sets comprising from-space and to-space have a large intersection, rather than being completely disjoint as in a pure copying collector.

Although we use a read barrier and a to-space invariant, our collector does not suffer from variations in mutator utilization because all of the work of finding and moving objects is performed by the collector.

Read barriers, especially when implemented in software, are frequently avoided because they are considered to be too costly. We have shown that an efficient read barrier implementation can be obtained using an optimizing compiler that is able to optimize the barriers.

We apply a number of optimizations to reduce the cost of read barriers, including standard optimizations like common subexpression elimination and specialized optimizations like barrier-sinking, in which the barrier is sunk down to its point of use, which allows the null-check required by the Java object dereference to be folded into the null-check required by the barrier (if the pointer might be null, the barrier cannot perform the forwarding unconditionally).

This optimization works with any null-checking approach used by the run-time system, whether via explicit comparisons or implicit traps on null dereferences. The important point is that we usually avoid introducing extra explicit checks for null, and we guarantee that any exception due to a null pointer occurs at the same place as it would have in the original program.

In the Jikes RVM implementation of Metronome, these optimizations resulted in fairly low read barrier overheads. On the SPECjvm98 benchmarks, the mean read barrier overhead is 4%, or 9.6% in the worst case (in the `201.compress` benchmark). Read barriers are not yet fully operational in our J9 implementation, but we will apply similar optimizations and expect to achieve similar results.

2.3 Time-Based Scheduling

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [3], has used work-based scheduling. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The reason for this is simple: work-based algorithms do a little bit of collection work each time the mutator allocates memory. The idea is that by keeping this interruption short, the work of collection will naturally be spread evenly throughout the application. Unfortunately, programs are not uniform in their allocation behavior over short time scales; rather, they are bursty. As a result, work-based strategies suffer from very poor mutator utilization during such bursts of allocation.

In fact, we showed both analytically and experimentally that work-based collectors are subject to these problems and that utilization often drops to 0 at real-time intervals.

Time-based scheduling simply interleaves the collector and the mutator on a fixed schedule. While there has been concern that time-based systems may be subject to space explosion, we have shown that in fact they are quite stable, and only require a small number of coarse parameters that describe the application's memory characteristics to function within well-controlled space bounds.

2.4 Provable Real-time Bounds

Our collector achieves guaranteed performance provided the application is correctly characterized by the user. In particular, the user must specify the maximum amount of simultaneously live data m as well as the peak allocation rate over the time interval of a garbage collection $a(\Delta GC)$. The collector is parameterized by its tracing rate R.

Given these characteristics of the mutator and the collector, the user then has the ability to tune the performance of the system using three inter-related parameters: total memory consumption s, minimum guaranteed CPU utilization u, and the resolution at which the utilization is calculated Δt.

The relationship between these parameters is shown graphically in Figure 3. The mutator is characterized by its allocation rate over the interval of a garbage collection $a(\Delta GC)$ and by its maximum memory requirement m. The collector is characterized by its collection rate R and a pre-selected fragmentation limit ρ (typically 1/8 worst-case fragmentation overhead is tolerated). The tunable parameters are Δt, the frequency at which the collector is scheduled, and either the CPU utilization level of the application u (in which case a memory size s is determined), or a memory size s which determines the utilization level u.

In either case both space and time bounds are guaranteed.

3. BETTER REAL-TIME COLLECTION

The Metronome system just described provides worst-case latencies of 3 milliseconds, suitable for the majority of real-time systems. However, there is still room for improvement and areas in which the technology needs to be extended.

3.1 Multiprocessor Real-time Collection

Large systems which track many concurrent events are often implemented on multiprocessors to achieve the desired level of performance and scale. The original Metronome algorithm was limited to uniprocessors, making it unavailable to this significant application domain.

The fundamental problem is that the Metronome collector relies on being able to make atomic (but small and bounded) changes to the heap during its execution quanta. This is accomplished by the use of *safe points*, which are inserted by the compiler into the application code at points where

certain invariants hold. A context switch into the collector may only take place when all threads are at safe points.

Safe points are essentially a low-overhead amortized synchronization technique. They allow the compiled code to perform heap operations without the use of expensive atomic operations or locking.

There are several kinds of synchronization between the mutators and the collector. When the mutator modifies the heap, it must inform the collector so that its tracing of the heap does not miss objects (via the write barrier). Similarly, when the collector moves an object (to reduce fragmentation and bound space consumption), it must inform the mutator so that it sees the new version of the object (via the read barrier).

To extend the Metronome algorithm to multiprocessors, there are basically two options: perform the work quanta synchronously across all processors, or design the algorithm so that the quanta can proceed concurrently with the mutators. Our current implementation uses the former approach, we call *stop the worldlet*.

Stop the worldlet has the advantage of (relative) simplicity in that basic atomicity constraints are still enforced. However, it is fundamentally limited in its ability to scale up to large numbers of processors, and in its ability to scale down pause times. This is due to the costs of barrier synchronization, unevenness in the work estimators, and other fine-grained load balancing issues. However, with careful engineeraing we are able to achieve very good results on multiprocessors of modest scale (so far, up to 8-way machines).

In addition to basic synchronization problems, multiprocessing also introduces some new issues specific to the real-time domain. In particular, the real-time guarantee must take into account the load balancing behavior across processors, since the collection does not complete until the last processor is finished.

At a theoretical level, this requires the programmer to bound another parameter, namely the longest chain of objects uniquely reachable from the roots [6]. The tracing of such a chain can not be parallelized, and therefore we must assume that in the worst case one processor begins processing the chain just as all of the others finish tracing the rest of the heap. In particular, this parameter determines the worst-case load balance.

The actual quality of load balancing is also determined by constant overheads and the trade-off between granularity and synchronization overhead; once again, good engineering can help stave off the inevitable. In practice, both load balancing problems become progressively more significant as the system is scaled up.

The extra time spent in load balancing does not adversely affect latency. It has two significant effects: the first is the delay in completion of a collection, which translates into a higher memory requirement (since the program will continue to allocate while it waits for the collector to finish). In practice, this does not appear to be a major problem on memory-rich multiprocessors.

The second significant effect is on utilization, since all processors are stopped synchronously. However, in some cases the work of various phases of collection can be overlapped, which reduces this effect. Further improvement can be obtained by allowing mutators to run concurrently with *some* phases of collection. In particular, the tracing phase, which is most subject to the fundamental load balancing problems described above, is amenable to execution in parallel with the mutators. The phases which are not amenable to parallelization with the mutators are stack scanning and defragmentation.

The implementation of Metronome in IBM's J9 virtual machine uses the stop-the-worldlet approach and is in use now by customers, who are achieving good results on 2- to 4-way multiprocessors.

In the long term, we seek to develop truly scalable algorithms. We have begun work on an almost wait-free algorithm, called Staccato, which will not only greatly increase the scalability of the collector but also allow further reduction in latency: the ability of mutators to run in parallel with any phase of the collector essentially allows extremely fast preemption.

3.2 Priorities, Latency, and Jitter

An issue that is often confusing to users is the question of the priority of the garbage collector. They want to set the priority of their real-time threads higher than that of the collector, so that they get serviced quickly even when collection is in progress. However, if this were done with operating system priorities that really were higher than those of the collector, then the collector could be interrupted while the heap was in an indeterminate state.

As a result, the collector runs at a higher "physical" priority (in the operating system) than Java threads that have access to the garbage collected heap. However, users may set threads to run at a higher "logical" priority than the collector, in which case the collector will voluntarily give up control as soon as it detects that a logical high-priority thread is ready to run.

For periodic threads and timers, since the collector knows the deadline in advance, it can adjust its work quantum in advance so that it finishes in time for the deadline. There is a small amount of jitter in the work estimator, but by factoring that jitter into the deadline we are able to meet time-based deadlines with a jitter of $\pm 3\mu s$. The cost is that we must spin during the over-provisioning period for the work estimator, which is typically $5\mu s$. Although there are occasional longer latencies, they are predictable and the collector can schedule around them.

However, the frequency of periodic threads and the latency for event-driven threads is still limited by the inherent latency of the collector.

3.3 Latency Reduction

The lower limit at which real-time garbage collection can be applied is determined by the worst-case latency of the system. This is currently about 2 milliseconds, making the system suitable for periodic tasks down to about 200 Hertz. While adequate for a large body of real-time systems, there are still many systems with lower latency requirements.

The worst-case latency is determined by the largest atomic step that the system needs to take. In a garbage collector, those steps typically consist of things like scanning the stacks for roots, scanning the global variables for roots, scanning the pointers of an object, moving an object, and so on.

In the Metronome, the use of arraylets ensures that both object scanning and object relocation are bounded and short operations. The use of a read barrier and mark-phase pointer fixup avoids the need for atomically updating all of the

pointers to a moved object. Global variables are write-barriered, so they need not be scanned for root pointers (the write barrier records them incrementally).

However, in our current implementation, two significant atomic steps remain: stack processing and finalizer processing. To achieve our current bounds, stack size and finalizer usage is limited. However, these are not fundamental problems and we are working to incrementalize them.

Stack processing can be incrementalized by the use of *stacklets*, which partitions the stack into fixed-size chunks which are then processed atomically [9]. This requires a modification to the call and return sequences so that stack overflow on call causes the insertion of a "return barrier", which snapshots the pointers of the stacklet below it before returning to the calling function.

With stacklets, the atomic root processing of the collector is limited to scanning the top stacklets of the currently running threads. We expect this to take in the neighborhood of 50 microseconds on current hardware.

The other problem is finalizer processing, or more precisely the processing of all of Java's "strange" pointers, which include not only unreachable finalized objects, but also weak, soft, and phantom references. This is not inherently difficult to incrementalize, but requires that all java.lang.ref types are implemented with a double indirection so that the requirement for atomic clearing can be met in unit time.

More problematic, however, is how to implement soft references, whose semantics almost require a stop-the-world garbage collection: " All soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an OutOfMemoryError". If this is done when memory is truly full, then by the time the collector makes its last-ditch attempt to reclaim memory, the real-time characteristics will have already been violated. The implementation is free to clear soft references at any time, and in our initial implementation we simply clear them on each collection.

However, a solution which retains the useful properties of soft references is desirable, but it is not clear how to reconcile this with real-time collection. In particular, since we require an upper bound m on the size of the live data in the heap, and soft references are explicitly designed to allow that quantity to be indeterminate. At present we have no solution to this problem.

3.4 Verifying Application Parameters

Metronome's ability to provide real-time guarantees is contingent on accurate characterization of the maximum live memory m and the maximum long-term allocation rate a. Clearly the ability to accurately characterize those quantities is essential to the correctness of the resulting system.

Allocation rate is the easier of the two to bound. In fact, bounding the allocation rate is simpler than computing the WCET of a task, since as Mann et al [19] have shown, it can usually be performed using an analysis that follows worst-case paths without knowing which ones are taken or how often. They achieved static bounds that were usually within a factor of two of the actual measured allocation rate.

In some cases it may be necessary to provide loop bounds to obtain a sufficiently tight bound on the allocation rate, but this is also true of WCET analysis.

The more difficult problem is computing the maximum live memory m. An accurate bound would essentially have to perform an abstract interpretation of the collector at compile-time, which in general will be unable to provide useful bounds for the kinds of complex programs of interest. Currently, for unrestricted programs, we do not see any alternative, except empirical methods based on test coverage.

However, it is important to distinguish the problems introduced by garbage collection *per se* from those introduced by the use of dynamic data structures which are inherent to complex real-time systems. The maximum live memory must also be computed in a system built with explicit allocation and de-allocation, or in a system using object pooling from a fixed-size pool, or in a system using scoped memory in all but the most trivial ways.

Fundamentally, the complexity of verification comes from the use of dynamic data structures. The additional complexity introduced by garbage collection is that the allocation rate must also be considered. However, this must be balanced against the reduction in complexity and the improvement in reliability provided by automatic garbage collection.

3.5 Verifying the Collector

The other aspect of correctness is that of the collector implementation itself. A real-time concurrent garbage collector is a very complex subsystem of the virtual machine, and the algorithms involved are notoriously difficult to prove correct. Since the collector now forms part of the trusted computing base of a critical system, verification becomes increasingly important.

The study of concurrent garbage collectors began with Steele [23], Dijkstra [11], and Lamport [17]. Concurrent collectors were immediately recognized as paradigmatic examples of the difficulty of constructing correct concurrent algorithms. Steele's algorithm contained an error which he subsequently corrected [24], and Dijkstra's algorithm contained an error discovered and corrected by Stenning and Woodger [11]. Furthermore, some correct algorithms [4] had informal proofs that were found to contain errors [20].

Many additional incremental and concurrent algorithms have been introduced over the last 30 years, but there has been very little experimental comparison of the algorithms and no formal study of their relative merits. While there is now a well-established "bag of tricks" for concurrent collectors, each algorithm composes them differently based on the intuition and experience of the designer. Since each algorithm is different, a correctness proof for one algorithm cannot be re-used for others.

Previous work on proving the correctness of concurrent collectors has applied the proof to the algorithm itself. Since the algorithm is complicated, the proof is as well, and therefore subject to error.

We have taken a different approach: rather than proving the ultimate algorithm correct, we start with an extremely simple algorithm which is amenable to a simple proof, and then transform it into a practical, efficient algorithm with a series of incremental transformations, each of which can also be proved correct [25, 26].

In the process of developing these transformations, we have generalized various aspects of concurrent collection, including the treatment of write barriers as reference counting operations, mixing incremental-update and snapshot approaches in a single collector, and providing a range of techniques for handling newly allocated objects.

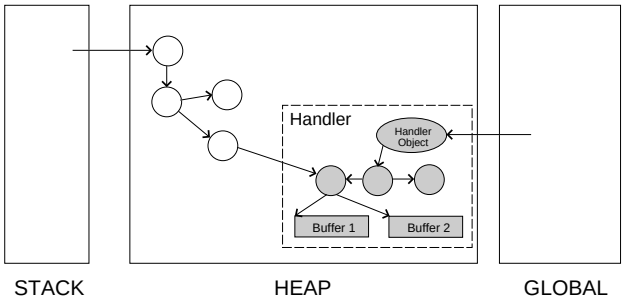


Figure 4: Interaction of Handlers with the heap. Handlers reside in the garbage collected heap, but are pinned (gray objects) during their lifetime. They may be referenced by other heap objects, and will be subject to garbage collection once the handler exits.

This has resulted not only in new insights, but also in the derivation of new algorithms, which have been shown both empirically [25] and formally [26] to be more precise than some previous algorithms. In particular, the new algorithm retains the predictable termination property of snapshot algorithms (which are necessary to achieve real-time guarantees) with the lower memory requirements of incremental-update algorithms.

We have formalized for the first time the notion of the *relative precision* of concurrent collectors, and express the transformations as correctness-preserving and precision-reducing. We observe that many precision-reducing transformations are also concurrency-increasing, and are currently working to formalize the notion of relative concurrency of collectors.

Ultimately, our goal is to produce the actual code of the collector via mechanical transformation from the simple, provable algorithm and a selection of provable transformations chosen to provide the desired performance properties.

4. SPECIALIZED CONSTRUCTS

Although real-time garbage collection is able to provide real-time behavior to extremely general code at a fairly high resolution, as discussed in Section 3.3 there appear to be fundamental limits on how low that latency can be driven.

Furthermore, in some situations, it is desirable to use a more restrictive programming model to enforce certain resource constraints or to improve the level of static verifiability of the system.

In this section we describe work in progress on the design and implementation of two such constructs, called *Handlers* and *E-Tasks*. Unlike the scoped memory construct of the Real-Time Specification for Java [7], these constructs are free of run-time exceptions and modularity-violating interface constraints. They are designed to fit the natural programming idioms of real-time systems, rather than being designed to avoid garbage collection.

Taken together, Handlers and E-Tasks are likely to entirely eliminate the need for low-level mechanisms like RTSJ's scoped and immortal memory, and replace them with safe, high-level constructs that are compatible with the standard Java language while requiring minimal changes to the virtual machine.

4.1 Handlers

Handlers are designed for tasks that require extremely low latencies and very high frequencies. They are based on the principle that as the frequency with which tasks are executed, their complexity of necessity drops.

Handlers are tasks that operate on a pre-allocated data structure whose pointers are immutable. The run-time system *pins* this data structure so that the garbage collector cannot move it, as shown in Figure 4.

Because Handlers cannot change the shape of the heap, and the garbage collector cannot change the location of the Handler's objects, Handlers can preempt the garbage collector at any time, even while the invariants of the rest of the heap are temporarily broken. This allows extremely fast context switching, limited only by the underlying hardware and operating system.

Rather than rely on dynamic checks, Handlers make use of Java's existing final mechanism, which is statically verified. However, Handlers have some additional restrictions that are checked at *instantiation time*, when the Handler is first loaded and before it is scheduled. A Handler contains a run() method and a set of local variables.

At instantiation time the validator checks that the data structure reachable from the local variables does not contain any non-final pointers, and that code reachable from the run() method does not access any non-final global pointers, manipulate threads, or perform any blocking synchronization operations.

If the Handler is valid, then the data structure reachable from its local variables is *pinned:* the garbage collector is informed that the objects are temporarily unmovable.

The Handler is now guaranteed to access only final pointers of pinned objects. Because the pointers are final, it will execute no write barriers and there is no mutator-to-collector synchronization; because the objects are pinned, the collector will not move them and there is no collector-to-mutator synchronization.

Handlers are well-suited to tasks that perform buffer processing. For instance, it may be necessary to sample a sensor at very high frequency. The sample data can then be processed by a lower-frequency task that analyzes the sample, perhaps performing convolutions and then choosing an actuator value. High-frequency buffered output can also be used to drive devices such as software sound generators, where the lower-frequency task can create a waveform and then pass it in a buffer to a Handler.

Figure 4 shows a canonical Handler that uses double-buffering. The buffers are exchanged between the Handler and the garbage collected tasks using non-blocking queues.

A prototype implementation shows that Handlers are likely to achieve latencies of a few microseconds on stock hardware and operating systems.

By comparison with RTSJ's scoped memory coupled with NoHeapRealTimeThread's [7], Handlers are both more reliable, since they throw no run-time memory access errors, and more efficient, since they require neither run-time checks nor run-time scope entry and exit (anecdotal evidence suggests that scope entry and exit costs about 16μs).

Handlers are more restrictive than scopes in that they do not allow dynamic memory allocation, but less restrictive in that they do allow references from the heap. The result is a more reliable, higher-performance programming construct that better matches programmer needs.

4.2 E-Tasks

Many real-time systems decompose naturally into a set of tasks that communicate solely by message passing. Such a decomposition provides a very high level of determinism and reliability because each task is purely functional in its inputs and the task abstraction matches the sensor-to-actuator control flow of many systems.

E-Tasks provide such a task model within Java, adapted from that of Giotto [13], but extending it to allow individually garbage collected tasks and the communication of arbitrarily complex values over ports.

Like Handlers, E-Tasks rely on instantiation-time validation. They share some of the same restrictions, but neither one is a subset of the other. E-Tasks are more restrictive in that they may not observe any global mutable state, nor may the heap contain references to objects inside of E-Tasks. On the other hand, E-Tasks are less restrictive in that they may allocate memory and mutate their pointer data structures.

Unlike Handlers, E-Tasks receive new objects from external sources (via ports). If those sources include types not previously seen by the E-Task, they could cause previously un-validated code to be executed in overridden methods. As a result, the E-Task validator must check not only its given task, but also ensure that any changes to the call graphs of preexisting tasks are benign.

E-Tasks and their ports also implement the *logical execution time* abstraction of Giotto, which provides platform-independent programming of the timing behavior of the task. The result is an extension of Java's principle of "write once, run anywhere" from the functional domain to the timing domain.

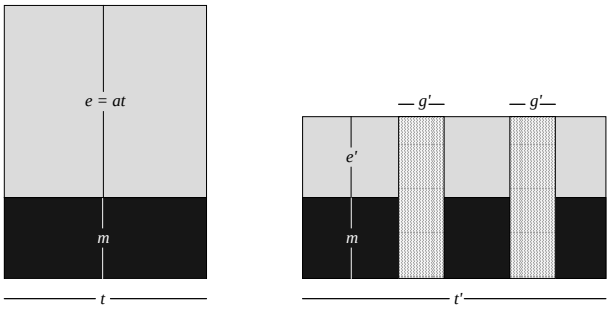
The logical execution time (LET) of a task determines how long the task executes in real time to produce its result, independently of any platform characteristics such as system utilization and CPU speed. Then, we check statically and dynamically if the task does indeed execute within its LET for a given platform, e.g., characterized by a scheduling strategy and WCETs. If the task needs less time, its output is buffered and only made available exactly when its LET has elapsed. As a result, the behavior of the system is both platform independent (assuming sufficient resources to complete on time) and composable (since two LET-based sets of tasks can be composed without changing their external behavior).

As we illustrated in Figure 3, the Metronome real-time collector allows a tradeoff between space and time based on the available CPU and memory resources. With E-Tasks, we can extend this notion to a finer granularity.

This is shown in Figure 5(a): we start by considering the execution of the E-Task in the absence of garbage collection. It runs for time t, has a base memory m of permanent data structures, and allocates memory at rate a. As a result the extra memory allocated by the task is $e = at$, and the total space required for the task is $s = m + at$.

However, if we wish to reduce the task's memory consumption we can interpose intermediate garbage collections which will temporarily reduce the memory utilization back to m. As a result, the task will require additional time (g' per collection) but consume less space, as shown in Figure 5(b).

The difference between a task's WCET and its deadline is called its *slack*. By analogy, we call the difference between a tasks base memory and its allocated memory its *slop*. Depending on the amount of slack and slop, garbage collections



(a) Original E-Task (b) Garbage-Collected E-Task

Figure 5: Trading Space for Time in Task Execution. Time is the horizontal axis, space is the vertical axis. m is the base memory of the task, t is its execution time, and a is its allocation rate.

can be introduced to trade space for time. Garbage collection is triggered by setting the limit of the private heap to some value between m and s; this is called its *space-out*.

The ability to make these space/time tradeoffs introduces the potential for sophisticated scheduling algorithms that consider not only time, but also space. Although they would, in general, be too expensive to perform online, it may be possible to compute them in advance and validate them with a witness in the manner of schedule-carrying code [14].

As Sagonas and Wilhelmsson have discussed in the context of Erlang (which has a similar task model) there are tradeoffs between using a global heap in which read-only objects are shared between tasks, private heaps which are individually collected, and a hybrid of the two [22]. E-Tasks present the same implementation choices; we concentrate on the use of private heaps for the level of accountability they provide, but there is nothing about the design that precludes the other approaches.

One of the major problems with the message-based model of E-Tasks is that it appears to require that data structures sent on ports be immutable; otherwise a data structure read from the same port by two tasks, or sent on a port and (perhaps partially) retained by reference in the task, would be subject to mutation that could cause side effects.

Such a restriction is undesirable because it would significantly limit the flexibility of the data structures and the ability to use pre-existing libraries to create and process them. However, it is not the mutability of the data structure that is the fundamental problem, but rather the potential for *sharing*.

We solve this problem with *send-by-collection:* at the end of the task execution, a specialized garbage collector copies the objects in ports from the sender to the receiver. In the event that it knows, either statically or dynamically, that there is no sharing, it may be possible to optimize that operation. However, if there is sharing, then multiple receivers will each receive their own copy of the mutable data, and there will be no side effects.

In addition to allowing the use of mutable data structures, send-by-collection means that E-Tasks can even make use of libraries that invoke synchronized methods because all data is guaranteed to be task-local and the synchronizations

are therefore guaranteed to be redundant. The specialized collection simply removes any synchronization state from the copied objects that are sent to other tasks.

5. ANALYSIS AND VISUALIZATION

Complex real-time systems are, well, *complex*. Therefore, it is crucial to be able to understand their behavior. The Metronome system supports this through an efficient, accurate trace facility and a visualizer called *TuningFork*.

5.1 Trace Generation

Although tracing tools are commonplace (albeit not as common as they should be), the generation and analysis of traces in a real-time system poses certain challenges.

Beginning with the earliest versions of the system we incorporated a cycle-accurate trace facility into the virtual machine. Trace events are usually fixed-size 16-byte records consisting of an 8-byte timestamp (cycle count) and an 8-byte data field.

The trace facility is designed to be efficient enough to be run in "always on" mode, although command line options are provided to disable it, and a build-time option allows us to generate a virtual machine that does not even contain the extra conditional tests.

The trace subsystem must be able to execute without interfering with the real-time behavior of the rest of the system. It therefore itself takes on many of the properties of a real-time system. Its operations must be bounded and lock-free. Therefore it may have to abandon buffers when the filesystem or socket is not draining the data quickly enough, or when the virtual machine is producing trace events too quickly.

When the system is extended to multiprocessors, the complexity of tracing increases considerably. First of all, most systems (with the exception of the IBM 390 [15]) do not have a globally synchronized high-resolution clock. In fact, there is both skew and drift. On a single board this is relatively low, since the processors typically share a single oscillator. However, across boards the effect can become considerable. Variations are caused by both static phenomena such as the use of different chips, and dynamic phenomena such as temperature variation.

While we perform a clock synchronization at startup and periodically piggyback clock synchronization on other synchronization events, there is always some level of uncertainty in the measurements.

As a result, any trace analysis or visualization facility must be prepared to deal with both incomplete and inaccurate data. This also places a premium on trace data that avoid dependence on previous entries. For instance, we initially recorded allocation and freeing with incremental amounts, but this led to incorrect results if even one record was lost, so we changed them to use absolute numbers. For events that are of necessity stateful, we include sequence numbers to allow the detection of dropped events and reconstruction of partial or estimated results.

The system is currently being extended to allow user-defined events and their insertion from Java code. This will allow users of the system to correlate application-level and virtual machine events. With suitable kernel extensions operating system events can be incorporated as well, allowing complete vertical profiling.

5.2 Trace Visualization: The Tuning Fork

The trace facility in the virtual machine provides the raw data, but it is also necessary to monitor and analyze that information. *TuningFork* is a trace visualization tool that is built as an Eclipse plug-in. TuningFork itself also exports a plug-in based architecture, so that the trace format itself is user-definable.

The various visualizations and analyses are also structured as plug-ins, allowing a great deal of flexibility. We are in the process of converting our off-line statistical analysis tools to the TuningFork architecture, and are investigating the possibility of allowing the same trace analysis plug-ins to be run inside of the virtual machine that is gathering the traces, so that it can be self-monitoring.

A screen shot of TuningFork is shown in Figure 6. It provides both time-series and histogram views of data, as well as a view of the ring buffer which contains chunks of data from the different streams that make up the trace.

In the future, we plan to add an oscilloscope-style view for visualizing very high-frequency events, and a heap density view in the style of GCspy [21].

As on the producer side, TuningFork must also be structured as a fault-tolerant real-time system. It must itself be prepared to discard buffers and to analyze data that is incomplete or not well-ordered.

In order to provide a unified view of the information from different parts of the system, the various streams (from different CPU's, threads, etc.) are merged into a single, sorted logical stream. There is a trade-off between the completeness of the stream and the timeliness with which it can be delivered; the system uses a paramterizable time window within which it must receive data from all streams before sorting and producing the result. This means that data arriving after the window will be discarded. In Figure 6, this is shown in the ring-buffer diagram in TuningFork's lower left pane, where the darkened set of buffers forms the "merge window".

Essentially, delay and loss are simply two sides of the same coin. They also occur at both the beginning and end of the merged in-memory events that can displayed by the visualizer, since when old buffers are discarded the data is lost in an order different from the sorted order.

6. PROBABILISTIC WCET

In all of our work on enabling the use of Java for programming complex real-time systems, a recurring theme is the fact that various assumptions are being made about the average behavior of the system.

This is true not only in the use of the long-term average allocation rate to derive a bound for real-time garbage collection, but is also implicit in the use of modern hardware with its numerous sources of non-determinism and unpredictability due to such things as caches, branch prediction, hyperthreading, and so on.

Traditional approaches to the design of real-time systems seek to maximize determinism to provide firm guarantees that tasks will complete within their deadlines. This approach is enshrined in the concept of worst-case execution time (WCET).

However, WCET analysis typically has to make many conservative assumptions, which leads to significant over-provisioning. With the current differential between cache

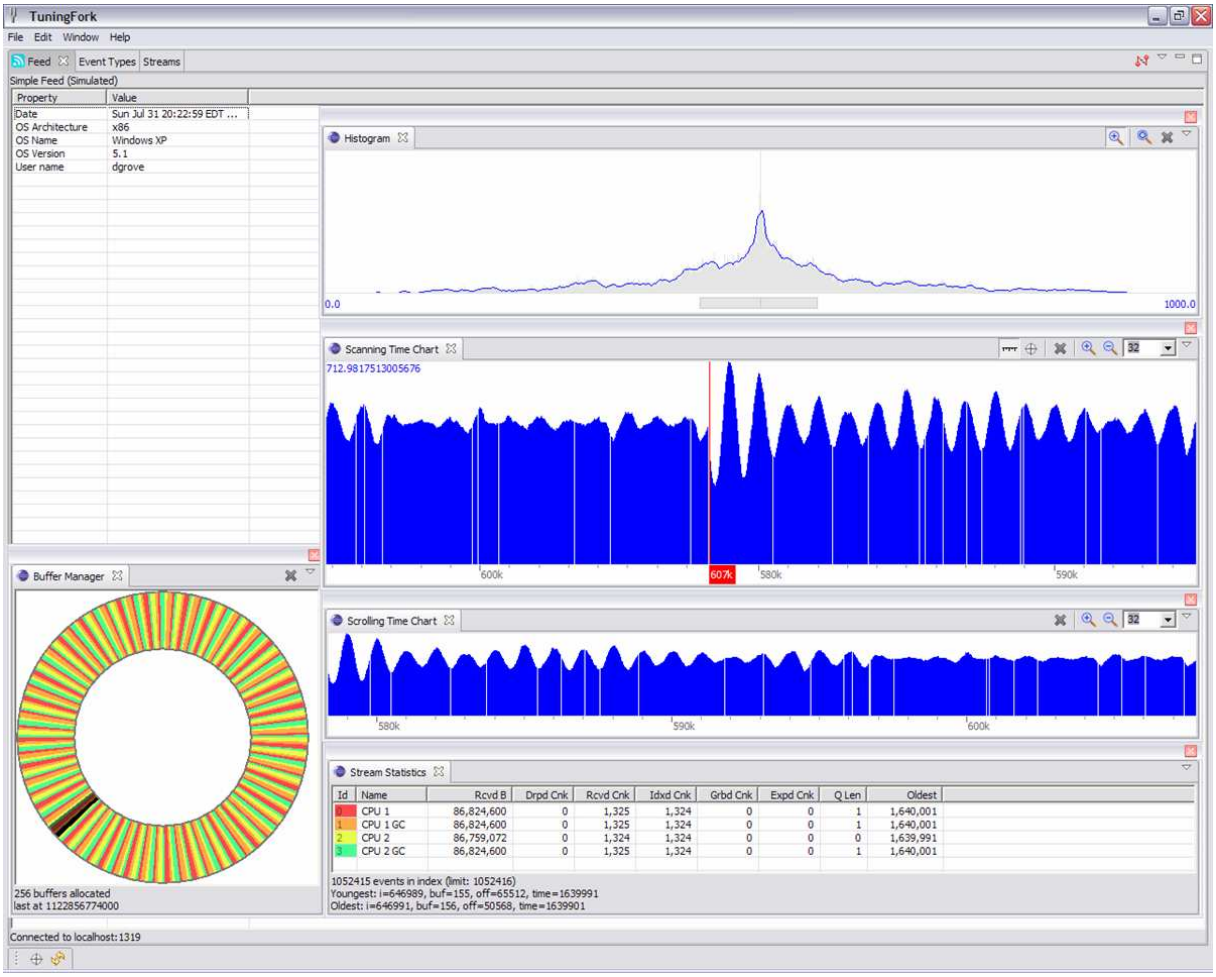


Figure 6: Screenshot of the Tuning Fork tool.

and main memory access, the level of over-provisioning is often so high as to be useless.

WCET also drives a design methodology, in both hardware and software, that places a strong emphasis on deterministic execution time of operations rather than on optimization for average-case execution time as is done in other branches of computer science.

With the new generation of complex real-time systems, the amount of variability increases and the WCET methodology does not scale up to the resulting levels of complexity.

We are investigating a different approach: real-time systems are composed of components in which there are many sources of nondeterminism. In fact, assuming that they are statistically independent, it is actually better to have *more* sources of nondeterminism rather than less. Our approach shares some features with that of Bernat et al [5], which applied such techniques to basic block profiles.

By using many statistically independent sources of nondeterminism, we can analytically determine the amount of over-provisioning required to meet an arbitrary level of confidence that the task will complete within the given time. Since the variance of a single type of event drops as the total number of events becomes large, and the variance of the composition of independent events drops super-linearly, the amount of over-provisioning required to reach extremely high levels of confidence is surprisingly small. Our approach is to replace a deterministic WCET with a bound whose probability of failure is below the mean time between failure (MTBF) of the physical components in which the real-time software is embedded. We call this Probabilistically Analyzed WCET, or PAWCET.

Our approach becomes more and more attractive as the number of non-deterministic operations increases. Thus it is well-suited for example to tasks with a 10 millisecond deadline running on a 1 GHz processor, where 10 milliseconds might comprise $10,000,000$ instructions, $2,000,000$ memory accesses, 5000 dynamic memory allocations, and so on. On the other hand, it is not so well suited for a 1 microsecond regime, or for a 10 millisecond regime on an 8-bit microcontroller running at 1 MHz.

However, more operation-rich regimes are exactly those to which we wish to apply our methodology, since very short-running programs are by their nature simpler to analyze using deterministic approaches (such as the technique of Ferdinand et al [12], which can tightly bound the costs due to cache, branch prediction, etc. for programs restricted to bounded loops over static data structures).

The probabilities can be analyzed using relatively recent results in the statistics of large deviations [10]. We have used these techniques to derive confidence formulae, and for determining the range of applicability of the formulae. In particular, for formulae that apply to a large number of events, we quantify what constitutes a "large" number. This number will be the inflection point below which traditional deterministic WCET methods must be used.

Our approach provides both a design methodology and a quantitative method of analysis. The design methodology is to make abundant use of optimizations for improving average-case performance, but to limit the variance of the worst-case performance, and to maximize the independence of the worst-case events from other optimized operations in the system.

The PAWCET analysis takes information about the number of nondeterministic events, their probabilities, and their degree of correlation, and provides an execution time estimate that will be met with a given degree of confidence.

By allowing a much wider scope for optimizations, the tasks will execute more quickly, which in itself makes it more likely that they will be able to meet their deadlines. Even more importantly it allows programmers to write simpler code, which will result in a corresponding increase in reliability.

Of course, the Achilles' heel of any statistical approach is unexpected correlation. PAWCET is only as good as the probability estimates upon which it is based. Thus another design principal is that systems should be designed to be resilient to correlation. For example, set-associative caches drastically reduce the execution time variance due to long-term correlations in cache misses.

While probabilistic techniques have their limitations, as real-time systems increase in complexity there will be no other viable approach. In the long term, we expect the methodology used to validate complex real-time systems will combine static analysis, measurement, and probabilistic analysis.

7. CONCLUSIONS

The growth in complexity of real-time systems has caused existing methodologies based around small, simple systems with totally deterministic behavior to break down. The situation is similar to that faced by hardware designers with the advent of VLSI some twenty-five years ago.

Spurred by the advent of real-time garbage collection, in conjunction with static compilation and the scheduling facilities of RTSJ, Java has reached a critical inflection point in its usability and credibility for the construction of large, complex real-time systems.

Continuing reduction in worst-case latency of garbage collection, coupled with increased scalability for multiprocessors, will cause the domain of applications not amenable to garbage collection to grow ever smaller.

For those applications with the shortest and most critical timing constraints, specialized constructs such as Handlers and E-Tasks promise to provide extremely low latency and very high predictability, while maintaining Java's high level of abstraction and its strong guarantees of safety and security.

The complexity of the systems involved means that absolute determinism is precluded by the undecidability of the analysis problems. Providing sophisticated tools for analysis and visualization allows the complexity to be understood, and principled probabilistic analysis allows the complexity to be controlled.

We believe that these and other advances will lead to the widespread adoption of Java for real-time programming in the coming years.

Acknowledgments

Much of this work was done in the IBM J9 virtual machine, and would not have been possible without the use of that infrastructure or the assistance of the J9 team, in particular Pat Dubroy, Mike Fulton, and Mark Stoodley. We also thank Bob Blainey, Tom Henzinger, En-Kuang Lung, and Greg Porpora for many useful discussions.

8. REFERENCES

[1] Bacon, D. F., Cheng, P., and Rajan, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, June 2003). *SIGPLAN Notices*, *38*, 7, 81–92.

[2] Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

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

[4] Ben-Ari, M. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst. 6*, 3 (1984), 333–344.

[5] Bernat, G., Colin, A., and Petters, S. M. WCET analysis of probabilistic hard real-time system. In *IEEE Real-Time Systems Symposium* (2002), pp. 279–288.

[6] Blelloch, G. E., and Cheng, P. On bounding time and space for multiprocessor garbage collection. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1999). *SIGPLAN Notices*, *34*, 5, 104–117.

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

[8] Brooks, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.

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

[10] Dembo, A., and Zeitouni, O. *Large Deviations: Techniques and Applications*, second ed., vol. 38 of *Stochastic Modelling and Applied Probability*. Springer-Verlag, 1998.

[11] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM 21*, 11 (1978), 966–975.

[12] Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R. Reliable and precise WCET determination for a real-life processor. In *Proc. of the First International Workshop on Embedded Software* (Tahoe City, California, Oct. 2001), T. A. Henzinger and C. M. Kirsch, Eds., vol. 2211 of *Lecture Notes in Computer Science*, pp. 469–485.

[13] Henzinger, T. A., Kirsch, C. M., and Horowitz, B. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE 91*, 1 (Jan. 2003), 84–99.

[14] Henzinger, T. A., Kirsch, C. M., and Matic, S. Schedule-carrying code. In *Proc. of the Third International Conference on Embedded Software* (Philadelphia, Pennsylvania, Oct. 2003), R. Alur and I. Lee, Eds., vol. 2855 of *Lecture Notes in Computer Science*, pp. 241–256.

[15] IBM Corporation. *Enterprise Systems Architecture/390 Principles of Operation*, ninth ed., June 2003.

[16] Jikes Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net.

[17] Lamport, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of the 1976 International Conference on Parallel Processing* (1976), pp. 50–54.

[18] Lee, E. A. What's ahead for embedded software? *Computer 33*, 9 (2000), 18–26.

[19] Mann, T., Deters, M., LeGrand, R., and Cytron, R. K. Static determination of allocation rates to support real-time garbage collection. In *Proc. of the ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems* (Chicago, Illinois, 2005), pp. 193–202.

[20] Pixley, C. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing 6*, 3 (Dec. 1988), 41–49.

[21] Printezis, T., and Jones, R. GCspy: an adaptable heap visualisation framework. In *Proc. of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, 2002), pp. 343–358.

[22] Sagonas, K., and Wilhelmsson, J. Message analysis-guided allocation and low-pause incremental garbage collection in a concurrent language. In *Proceedings of the Fourth International Symposium on Memory Management* (Vancouver, British Columbia, 2004), pp. 1–12.

[23] Steele, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.

[24] Steele, G. L. Corrigendum: Multiprocessing compactifying garbage collection. *Commun. ACM 19*, 6 (June 1976), 354.

[25] Vechev, M. T., Bacon, D. F., Cheng, P., and Grove, D. Derivation and evaluation of concurrent collectors. In *Proceedings of the Nineteenth European Conference on Object-Oriented Programming* (Glasgow, Scotland, July 2005), A. Black, Ed., Lecture Notes in Computer Science, Springer-Verlag.

[26] Vechev, M. T., Yahav, E., and Bacon, D. F. Parametric generation of concurrent collection algorithms. Submitted for publication, July 2005.

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