RC24272 (W0705-143) May 30, 2007
Computer Science

IBM Research Report

Design and Implementation of a Comprehensive Real-time Java Virtual Machine

Joshua Auerbach[1], David F. Bacon[1], Bob Blainey[2], Perry Cheng[1], Michael Dawson[2], Mike Fulton[2], David Grove[1], Darren Hart[3], Mark Stoodley[2]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[2]IBM Software Group

[3]IBM Linux Technology Center

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Design and Implementation of a Comprehensive Real-time Java Virtual Machine

Joshua Auerbach[⋆]
Perry Cheng[⋆]
David Grove[⋆]
[⋆]IBM Research

David F. Bacon[⋆]
Michael Dawson[‡]
Darren Hart[†]
[†]IBM Linux Technology Center

Bob Blainey[‡]
Mike Fulton[‡]
Mark Stoodley[‡]
[‡]IBM Software Group

ABSTRACT

The emergence of standards for programming real-time systems in Java has encouraged many developers to consider its use for systems previously only built using C, Ada, or assembly language. However, the RTSJ standard in isolation leaves many important problems unaddressed, and suffers from some serious problems in usability and safety.

As a result, the use of Java for real-time programming has continued to be viewed as risky and adoption has been slow.

In this paper we provide a description of IBM's new real-time Java virtual machine product, which combines Metronome real-time garbage collection, ahead-of-time compilation, and a complete implementation of the RTSJ standard, running on top of a custom real-time multiprocessor Linux kernel.

We will describe the implementation of each of these components, including how they interacted both positively and negatively, and the extensions to previous work required to move it from research prototype to a system implementing the complete semantics of the Java language. The system has been adopted for hard real-time development of naval weapons systems and soft real-time telecommunications servers. We present measurements showing that the system is able to provide sub-millisecond worst-case garbage collection latencies, 50 microsecond Linux scheduling accuracy, and eliminate non-determinism due to JIT compilation.

1. INTRODUCTION

Real-time systems have reached a critical inflection point in both their breadth of use and the complexity of their design. The small stand-alone real-time applications of the past are giving way to a new type of networked real-time system incorporating many individual components across numerous machines, whose timeliness requirements range from microseconds to seconds, and whose aggregate size is in the tens of millions of lines of code. Examples of such systems include integrated battleship management, VOIP-based telecommunications systems, stock arbitrage, and automotive systems.

Because of the size, complexity, and longevity of these systems, there has been increasing desire (and pressure) to implement them in Java. However, until now this had not been practical.

Some pieces to the puzzle existed: our Metronome system [5] demonstrated that it was possible to build a truly real-time garbage collector, eliminating the largest obstacle to using standard Java code for real-time programming. The Real-Time Specification for Java (RTSJ) standard [8] provided much-needed real-time scheduling APIs. Unfortunately it incorporated non-standard memory management (Scopes) and was oriented towards J2ME-based uniprocessor embedded devices. The RTSJ also left key issues such as class loading and JIT compilation completely unaddressed.

In this paper we describe our design and implementation of the first production Java virtual machine to solve these issues in a comprehensive manner. The IBM Real-time Java virtual machine combines

- real-time garbage collection,
- ahead-of-time compilation,
- a complete implementation of the RTSJ standard,
- pre-loading of classes,
- customized standard libraries,
- sub-microsecond vertical instrumentation,
- full J2SE support,
- multiprocessor capability, and
- co-existence of multiple real-time JVMs on a single host.

It runs in hard real-time mode on top of a real-time Linux kernel developed by IBM and the open source community, and in soft real-time mode on top of a standard Linux kernel.

The integration for the first time of all of these pieces of real-time technology into a single system provided unique challenges and required considerable extensions and adaptations of existing techniques.

Because of the large number of components, we will of necessity be unable to treat any one of them in detail. Instead we will focus on the novel aspects of the system which were required to make all of the different components work together, and to have each piece of technology implement the complete semantics of both the complete Java 5 language as well as the complete RTSJ specification. This includes the aspects of the underlying operating system that were developed collaboratively (with the IBM Linux Technology Center) to provide the necessary features for the JVM.

The cumulative effects are shown in an experimental evaluation of the system across a number of different benchmarks.

The resulting product has been adopted by Raytheon to produce the software for DDG 1000, the next-generation U.S. Navy destroyer. It is also being used by others in the government, financial, and telecommunications sectors.

2. THE J9 VIRTUAL MACHINE

The IBM Java Virtual Machine (JVM) is called J9. The core components in the JVM are the Interpreter, Just-In-Time (JIT) compiler, garbage collector, class loader and stack walker. There are interfaces between all of these components. For example, when a class is loaded, the class loader updates structures used by the garbage collector, interpreter, and JIT compiler, as well as code generated by the compiler. Similarly, when the garbage collector

unloads a class, the JIT is notified so that it can update it's internal structures. The code generated by the JIT is dependent on the type of garbage collector that is present. The garbage collector is pluggable, namely the garbage collector used by the JVM is loaded at startup based on the command-line options specified by the application programmer.

The J9 JVM is designed to be portable, and runs on dozens of variants of operating systems and hardware combinations. The way we can achieve this high level of portability is through a very small *portability layer*. This portability layer provides a mapping from the general services needed by the JVM, such as acquiring memory, writing files, etc., to the underlying operating system and hardware.

The Real Time JVM is built using this same philosophy. We introduce a new garbage collector called Metronome, along with a new set of interfaces that the JIT and interpreter need to implement. We extend some of the core services to support the RTSJ class libraries, in particular, priority threads, scopes and immortal memory areas, and we modify our optimization strategy to support Ahead-of-Time compilation along with conservative code generation that is optimized for predictable performance instead of strictly throughput performance.

3. REAL-TIME LINUX

IBM Real-Time Java runs in two modes: soft real-time on a stock Linux kernel and hard real-time on a real-time Linux kernel. In order to provide a Linux kernel which met the real-time requirements of the RTSJ and the Real-Time JVM, the IBM Linux Technology Center (LTC) developed a customized kernel in close association with the developers of the Real-Time Java virtual machine, and the open source Linux community. The kernel modifications are available as open source [19].

While the mainline Linux OS provided some real-time features such as the `SCHED_FIFO` scheduler policy and partial kernel preemption, it required changes to three key areas before it could provide the capabilities required by the real-time JVM to support the Real Time Specification for Java (RTSJ) and Metronome garbage collection. The community driven `PREEMPT_RT` patch provided much of the needed functionality, but was in need of some refinement and hardening. The current real-time Linux OS provided by the Linux Technology Center consists of Linux 2.6.16 and the 2.6.16-rt22 `PREEMPT_RT` patch, with additional bug fixes, hardware enablement patches, and RTSJ compliance features.

3.1 Locking

Locking mechanisms are a prime source of latencies. Latencies can arise due simply to lock contention, the need to enter kernel space to acquire frequently accessed locks, or from something more critical, like a priority inversion. The mainline kernel used spinlocks internally for several structures. The real-time kernel has replaced nearly all kernel spinlocks with mutexes (or sleeping spinlocks) which allow what would otherwise be a busy spin loop to be preempted to allow needed work to get done. Early in the development cycle, the LTC worked closely with the community to help guide a third generation fast userspace mutex implementation. This enabled the acquisition of uncontended locks to occur completely in userspace, reducing the number of times a task must enter the kernel. Lastly, priority-inheritance-enabled mutexes ensure high priority tasks block on needed locks for as little time as possible.

3.2 Scheduling

Even with a fully preemptable kernel, scheduling real-time tasks on an SMP system is a difficult problem. On a uniprocessor system, there is only one scheduler run queue, and selecting the next highest priority task to run is trivial. For performance reasons, the Linux scheduler maintains a separate run queue per CPU, which reduces lock contention and takes advantage of cache locality. When scheduling real-time tasks however, it is required that the *N* CPUs on the systems be running the *N* highest priority runnable real-time tasks at any given point in time. The LTC authored and published test cases that highlighted problems in scheduling corner cases that have since been resolved and incorporated back into the `PREEMPT_RT` patch. When a new real-time task becomes runnable and preempts the task running on the current CPU, the priority of the running tasks on all the remaining CPUs must be evaluated to see if the preempted task should preempt one of them. While costly, this approach ensures that strict real-time scheduling is preserved, even on an SMP system.

Interrupt handlers also contribute significantly to system latency. They typically run in kernel context, preventing anything else from using the CPU executing the handler. The real-time kernel converts the handlers to threads which can be prioritized and scheduled as a regular process. This approach provides ultimate priority control to the system designer, allowing them to specify a critical thread to run as needed even at the expense of the interrupt handlers.

Traditionally, tasks blocked on a lock or a condition variable would be woken up and scheduled in FIFO order. In a real-time system, the tasks' relative priorities must be considered as well, and given more weight. The real-time kernel wakes tasks from these states first by priority, and then in FIFO order.

3.3 Time and Timers

Early 2.6 Linux kernels relied on the periodic timer tick to track time. The timer tick is an interrupt handler that would increment jiffies by one unit somewhere between 250 and 1000 times a second, depending on the configuration of the running kernel. Time was then calculated by multiplying the jiffies value by the expected duration of a jiffy and then interpolating time that had elapsed since the last tick. This method was error prone, subject to missed interrupts, variations in the system clock frequency, and abrupt NTP adjustments. Time would at times even appear to go backwards!

Real-time applications require highly accurate timekeeping in order to schedule events and track their own deadlines, which led us to rewrite the kernel timekeeping code, now also present in recent mainline kernels [25]. Most of the actual timekeeping code is moved into generic code, requiring only a small amount of architecture-specific code to define a counter, frequency, and value ranking for each *clocksource*, enabling the system to select the best available clocksource at boot time. This *clocksource* abstraction replaces the timer tick as the underlying mechanism for measuring time. Not only does this new approach avoid error-prone interpolation, but it also allows for smooth frequency-based NTP adjustments.

In parallel with the development of the new generic timekeeping subsystem, Gleixner et al [17] authored a new subsystem with support for high-resolution timers (`hrtimers`). This new system makes a distinction between timers (which are expected to expire) and timeouts (which typically only expire in an error condition). Timeouts don't require higher resolution and continue to use the traditional tick-granular Linux timer wheel. New nanosecond-resolution `hrtimers` were separated from the traditional timer system and stored in a more appropriate time-ordered red-black-tree, to be processed independently from the timer tick.

Finally, a new clock events system was added to manage the various global and per-CPU interrupt sources. Similar to the clocksource, these clock event devices are small architecture-specific drivers that are managed by architecture generic code. By combin-

ing high-resolution timekeeping with high-resolution timer structures, the clock events are used to fire interrupts as needed to expire the hrtimers. This provides lower latencies without affecting the existing periodic tick and timeout infrastructure.

3.4 Configuration

Each real-time system is unique. Innumerable hardware configurations and workloads require that each system be at least partially tailored to the purpose for which it is destined. At the same time, any successful product must function out of the box. To this end, the LTC authored several custom configuration utilities which provide a sane default installation, but allow for individual customization.

The `set_kthread_prio` script, for example, runs at boot time and configures the priority of each kernel thread (such as the new threaded interrupt handlers) according to the values specified in a human readable configuration file. These priorities can be easily modified to suit the needs of a particular application or customer. As an example, certain applications may be dependent on efficient handling of network packets, while others may use the network only for routine tasks. By adjusting the relative priority of the networking kernel threads, each system can achieve real-time behavior where it matters, without patching or reconfiguring the Linux kernel.

In a standard Linux installation, only the root user has the necessary privileges to set thread scheduling policies and priorities. In order to allow real-time Java to be executed in user mode, a patched version of the PAM library was created that allows access to these real-time features to any user that is a member of the `realtime` group.

4. REAL-TIME GARBAGE COLLECTION

Metronome [5] is a hard real-time garbage collector that provides highly predictable latencies down to the millisecond level. The technology is different from earlier approaches to "real-time" collection in several fundamental ways.

First, Metronome's approach to collection allows virtually all of the collection work to be done asynchronously by the collector. This is in contrast to previous approaches, which often required the application threads to perform work (like copying objects or updating pointers) on behalf of the collector, causing uneven and unpredictable performance of the application threads.

Second, Metronome takes advantage of this decoupling and uses *time-based* scheduling of garbage collector work, where the collector (when it is active) runs at regularly scheduled intervals of regular sizes. This makes the impact of the collector on the application highly predictable and reliable. By contrast, previous approaches typically performed *work-based* collection, where a unit of collector work was performed in response to application activity, for instance for every 4KB of memory allocated. Since allocation is typically unevenly distributed, work-based scheduling results in uneven interruptions of the application.

Third, Metronome is engineered in such a way that its individual work quanta are extremely small, both in the average and worst case, to the point where its resolution is sufficiently high for the vast majority of real-time applications.

Metronome was originally implemented as a research prototype in the Jikes RVM [1] virtual machine at IBM Research. In that system, we only addressed real-time garbage collection, but not the other features required for a complete system solution, that are described in the subsequent sections of this paper. It also did not implement certain Java language features, like weak/soft/phantom references. Most importantly, it was designed for uniprocessor systems with the assumption that the real-time application would have full control of the CPU.

In the process of implementing a production-quality Metronome collector, we discovered a number of issues that arise from supporting the complete language semantics and from interaction with RTSJ features. We also made some significant improvements to the original system, supporting SMPs and driving down worst-case latencies by another order of magnitude.

4.1 Overclocked Scheduling

Metronome's time based scheduler operates by providing a *minimum mutator utilization* or MMU [11]. The MMU specifies a time window and a percentage of the CPU time within that window which must be allocated to the application (a.k.a. the mutator). For instance, an MMU of 70% with a 10 ms time window means that for *any* 10 ms time period in the program's execution, the application will receive *at least* 70% of the CPU time (i.e., at least 7 ms).

The original Metronome scheduler worked by always trying to perform a collection quantum that was exactly the size allocated to it in the MMU time window (using the example above, 3 ms). This led to two problems: first of all, those quanta were large enough that they caused uneven distributions for certain kinds of operations, and second, if the collector made mistakes in its estimate of how long a piece of work would take, it would overshoot its quantum size, and violate its MMU contract. This then required compensation by shortening the collector quantum to include the jitter, leading to a slowdown of the collection process.

In the production system, we solved these problems by *overclocking* the collector scheduler. The collector runs at a nominal quantum size of 500 μs, and those quanta are spread out over the MMU window. Thus for the example above, we would nominally perform 6 separate quanta to perform the 3 ms of collector work. Those quanta are evenly spread, so that the application experiences less and more evenly distributed collector-induced jitter. Furthermore, the scheduler is able to do a much better job of guaranteeing the MMU target, since if a single quantum is larger than expected, it can simply either delay the next quantum slightly, or else perform a shorter quantum the next time.

4.2 Stop-the-Worldlet SMP Scheduling

Metronome was originally designed for a uniprocessor, since we expected it to be used primarily for small embedded systems. This turned out to be advantageous, because it forced us to solve the real-time scheduling problems without relying on an extra processor (which might not have been properly provisioned).

To our surprise, the early customers for the technology were building real-time systems on blade-based 4-way SMP servers. This necessitated adapting the technology significantly. However, because our fundamental real-time scheduling approach was sound, we were able to apply it in an SMP environment as well: all processors simply perform their collector quanta in a synchronized manner.

For the relatively small multiprocessors we targeted, the overhead of barrier synchronization is sufficiently small that it does not appreciably impact the amount of collector work that can be performed.

In order to scale properly we then had to adapt the collection algorithm to operate in parallel, especially for the long phases such as marking live data and sweeping free objects. While this constituted a significant amount of engineering work, we were able to employ mostly previously known techniques.

4.3 Root Processing

The original Metronome prototype scanned all roots in a single collector quantum in order to obtain a correct initial snapshot. Otherwise, the assumptions of its snapshot-at-the-beginning approach would have been violated. In the production implementation root processing is considerably more complex because of the need to support very large numbers of threads and to implement the full Java language semantics which include several special types of pointers.

In making the scanning of threads incremental, our goal was to deal with large numbers of threads rather than to support very deep stacks in a single thread. Proposals to deal with the latter problem exist [12], but for the expected workloads it was necessary and sufficient in practice to scan some integral number of threads in each collector quantum. To support scanning only a subset of the threads in each quantum, we employed a conservative (overinclusive) approximation of snapshot called *fuzzy snapshot*. It works as follows.

In the very first quantum of a garbage collection, root processing begins. In a program with a small number of threads, it may complete within that first quantum. In this case, a standard Yuasa write barrier [26], which causes all overwritten object references to be remembered, and remains in effect until the end of the tracing phase of collection. Thus, the scanned threads and the mutable heap state (including class statics) are part of the snapshot.

In the event that it is not possible to scan the roots of all threads in the first quantum, an additional write barrier is enabled for those threads. This barrier records object references that are stored rather than the references that they overwrite (which are recorded by the Yuasa barrier), and is turned off individually for each thread once its roots are scanned.

The second barrier ensures that no as-yet-unscanned root can pass into an already-scanned thread and thereafter become the sole path to a live object. It works because all objects must pass through some mutable heap location to reach another thread. We call this combined barrier the *double barrier* because it remembers both overwritten and newly stored pointers for unscanned threads (as well as just overwritten ones for scanned threads).

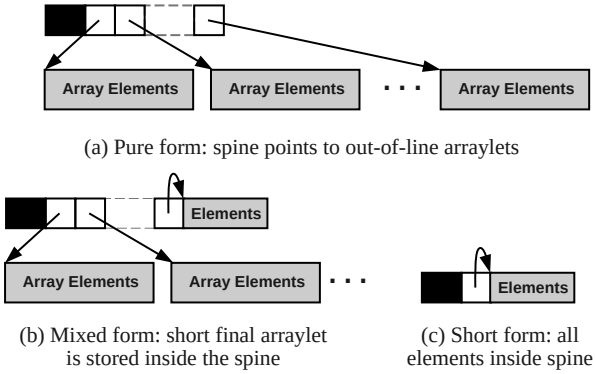
A fuzzy snapshot is a safe approximation of a snapshot even though it may omit roots that the snapshot would have included. These lost roots are precisely the ones that the unscanned threads discarded before they could either be scanned or recorded by the double barrier. But, since these roots no longer exist in the original thread and were never exposed to any other thread, they cannot be the sole means of keeping any objects live and so do not need to be roots.

4.3.1 Other Roots

The original Metronome implementation, like many other implementations of low-latency collectors, assumed that roots consist only of stack variables and static variables. The full Java language has several additional categories of roots, all of which have specific data structures and access patterns in the language, and some of which (finalizable objects, soft references, weak references) imply special semantics that the garbage collector must impose.

Each of these types of roots must be handled individually in terms of (1) how they must behave in relation to the double barrier, (2) how to incrementalize processing of the data structures that contain them, and (3) for the roots that have special semantics, how to preserve those special semantics while processing the roots incrementally. In practice, issue (2) is pure engineering and we do not go into detail about it here.

As a relatively simple example of the first issue, consider JNI global references. These may be created and destroyed through the



(a) Pure form: spine points to out-of-line arraylets

(b) Mixed form: short final arraylet is stored inside the spine

(c) Short form: all elements inside spine

Figure 1: Supported variants of arraylet object model

action of JNI functions but are never overwritten in place. Despite that fact, we must model both creation and destruction as write operations: conceptually, the former replaces null with non-null and the latter does the opposite. Thus, the double barrier applies to these operations. JNI reference creation remembers the just-created root if the operation is performed by an unscanned thread while the barrier is active. JNI reference destruction remembers the just-destroyed root if it happens when the barrier is active.

Soft and weak references present much more complex problems. These "roots" are only scanned after all objects reachable from the starting root set have been marked. Java requires that if a weak reference points to an unmarked object, then the weak reference should be cleared. For soft references, the virtual machine is expected to clear a subset of the references in the event that there is memory pressure (but the remaining soft references and anything reachable from them must be marked live).

Because soft and weak references are scanned late, the Reference.get() method is intercepted by the virtual machine with what amounts to a special-purpose barrier. During a collection, up until clearable reference roots are scanned, any object that is returned by this method is also remembered as a root.

When the time comes to scan the clearable references, the soft references are scanned twice. In the first scan, a subset of the soft references are chosen for preservation. These become roots in a continuation of tracing (i.e., all objects reachable from those roots are marked). Because Java has a consistency requirement for soft reference clearing, it is not sufficient just to mark the immediate referents.

After the first scan of soft references, we *commit* the set of live reference types by setting a global flag indicating that marking of special reference types has completed. The Reference.get() operation has been modified to return null when this flag is set *and* the underlying reference has not been marked by the collector. In this manner, we avoid the need to clear all of the references atomically, and are able to incrementalize clearing. In the second scan of soft references and the sole scan of weak references, those references that point to unmarked objects are actually cleared, after which the bit may be reset.

4.4 Arraylets

In order to bound the work of scanning or copying an array (one aspect of bounding collector pauses), and also to bound external fragmentation caused by large objects, Metronome relies on *arraylets*, a technique for breaking large arrays into chunks that can be processed in bounded time. The implementation of arraylets in a production JVM was one of the more difficult implementation

tasks that we faced.

In the arraylet object model, all arrays consist of two parts: a *spine*, which contains the object header and pointers to *arraylets*, which are contiguous fixed-size chunks[1] of memory that contain the array's data elements. If the last arraylet is smaller than the arraylet chunk size minus the spine header size, then it is allocated contiguously with the spine itself. This leads to three basic array organizations, as shown in Figure 1. Unlike a traditional array object model, with arraylets it is not the case that the data portion of an array is a single contiguous piece of memory.

In addition to the obvious changes in the interpreter and JIT compiler to perform the extra indexing calculations and indirection for Java-level array access, a number of other parts of the JVM and class libraries needed to be changed as well. Array manipulation, especially manipulation of `byte` and `char` arrays is ubiquitous, and the assumption that array data was always contiguous had influenced the design of many internal APIs.

Arraylets also interacted with user and class library native code. Although the JNI functions `GetArrayElements` and `GetPrimitiveArrayCritical` include an `isCopy` parameter to indicate whether or not the JVM has returned a direct pointer or a copy of the argument arrays data, it is somewhat common for programmers to write incorrect JNI functions that do not check this value. We discovered several such bugs in our own class libraries.

Finally, and unlike the JNI API, the two array-related functions in `sun.misc.Unsafe`, `arrayBaseOffset` and `arrayIndexScale`, implicitly assume contiguous arrays. To adapt this class to arraylets, we allowed the user to compute array "offsets" as if the array data was contiguous but then in the various native methods that used the offsets on array objects we first reconstruct the actual index and then do the arraylet indexing calculation. This is a suboptimal solution, but the best option available within the flaws of the established API.

5. RTSJ AND REAL-TIME COLLECTION

5.1 NHRT null Stores

The goal of RTSJ's No-Heap Real-Time Threads (NHRTs) is to provide a restricted programming model in which tasks can be written that can be run without interaction with or interference from the JVM's garbage collector. The NHRTs and the GC are intended to be completely decoupled. However, in the process of implementing NHRTs and Metronome, we determined that the existing specification of NHRTs has a somewhat simplistic view of NHRT/GC interactions based on stop-the-world garbage collection algorithms.

In particular, although NHRTs are not allowed to read any heap references, they are allowed to overwrite memory locations (in class statics, immortal memory area, etc.) that contain heap references. This impacts the root set used for garbage collection. In the context of an incremental collector such as Metronome, the act of overwriting such a pointer incurs an obligation to notify the collector that a root has been destroyed by adding it to a write buffer. To preserve the integrity of the write buffer data structure, when the collector is removing entries from the buffer it holds a mutex. If an NHRT needs to add an entry to the buffer, it must acquire the mutex and may be delayed until the GC finishes manipulating the write buffer. This window of potential interaction can be made fairly small, and only applies to NHRTs which actually overwrite heap references, which is arguably not a common operation in well-written NHRT code. However, given that the specification allows NHRTs to overwrite heap references, we do not believe it is possible to completely avoid NHRT/collector interactions except under fairly simple GC algorithms (which are not well suited to real-time JVMs).

[1]Using 2^n sized chunks allows the indexing calculation to be done with shift and mask operations. WRT currently uses 2 KB chunks.

5.2 GC Priority and Thread Priorities

Like many other garbage collection implementation, Metronome utilizes dedicated GC worker threads to perform collection work. Our real-time Linux implementation uses priority-based scheduling. Therefore, the priorities at which the GC worker threads are allowed to run must be carefully chosen to achieve the desired scheduling of GC activities both with respect to other threads within the JVM and with respect to other JVMs and other real-time processes running on the machine.

The main constraints are: (1) GC threads must run at a strictly lower priority than NHRTs to prevent them from interfering with an NHRTs execution, (2) GC threads must run at a priority at least as high as any heap-allocating thread (both normal Thread and Real-time Thread) in the same JVM to prevent them from being starved by the application threads whose heap allocation they are supporting and (3) it must be possible to run multiple real-time JVMs on a single processor and reason about whole-system scheduling properties.

We believe the best way to solve this design problem is to dynamically vary the priority of each JVM's GC worker threads such that they are marginally higher than the priority of any active heap-allocating thread within that JVM. In effect, this makes the GC threads (and this the JVM) have the same relative priority to other processes on the system as the realtime threads within in JVM. When mapping RTSJ priorities onto Linux priorities, we leave gaps such alternating OS priorities are reserved for Java threads and GC worker threads respectively.

Although assigning priorities in a system comprising multiple JVMs is still a complicated task, we believe that by having the JVM dynamically match GC thread priorities to that of the heap-allocating threads within the JVM we have at least not made the problem any harder.

5.3 Dual Barriers

Both Metronome and RTSJ rely on read and write barriers for correct operation. These barriers represent checks and actions that must be performed before and/or after reads or writes of object fields, array elements, and global roots. In the realtime JVM, we must execute a combined barrier that performs all of the operations of the two logically distinct barriers in an appropriate order. The resulting barrier sequences are complex enough that it is infeasible to inline the entire sequence everywhere; instead just the most common short-circuit operations are done inline. A detailed discussion of the barrier sequences and some of our initial work on optimizing the common cases can be found in [16].

6. RTSJ AND CLASS LIBRARIES

The addition of RTSJ into a Java VM adds several complications for the standard class libraries. This is because the existing class libraries were not designed to accommodate the new RTSJ concepts. Some of the specific areas in which issues were encountered during our implementation were global objects, caching, lazy initialization and consumption of Immortal memory.

6.1 Global objects

The class libraries provide direct or indirect access to a number of objects which are accessible to all threads within the vm. Examples include class objects, Security Managers, system properties,

annotations, the default ThreadGroup and so on. Since NHRTs cannot access objects allocated from the Heap, additional work is required within the class libraries to ensure that these objects are either allocated in Immortal so that they can be directly accessed, or that there is some other way to provide the equivalent in a transparent manner. While the RTSJ identifies some of these objects (for example class objects and system properties) and mandates that they be allocated from immortal, we found that there are others that had to be identified and addressed. To identify these issues systematically we started with an analysis of static fields within our class library objects and the set of objects accessible through them.

6.2 Lazy Initialization

In order to conserve resources, the class libraries sometimes defer initialization/creation of objects until the first time they are required. This can cause problems once RTSJ support has been added, as the current allocation context when objects are initialized/created may not be suitable. This lazy initialization can result in global objects being created from the heap such that they can later not be accessed by NHRTs, failure to store the object created if the current allocation context is a scope and the object would be stored into an object allocated from Immortal, and so on. We had to adjust a number of places within the class libraries to move the initialization/creation so that it occurred during vm initialization or so that if lazy initialization was used,the objects were created in the appropriate memory area regardless of the current allocation context.

6.3 Caching

Caching is used in the class libraries to reduce resource usage and to enhance performance. We saw several different problem with respect to caches. The first had to do with "polluted objects" in which a heap object was added to a cache by a non-NHRT and subsequently NHRTs would get a MemoryAccessError trying to use the cache. In this case NHRT access to the cache might or might not succeed depending on the objects that it needed to access from the cache. The second case was related to "polluted containers" which occurred when the expansion or reallocation of internal containers for the cache occurred in heap and all subsequent attempts by an NHRT to use the cache would fail with a MemoryAccessError. Variations of these two problems occurred when the current allocation context was a scope and IllegalStoreExceptions occurred when new objects were being added to the cache or the internal containers for the cache were being reallocated. It was necessary to either modify the operation of the caches to be safe regardless of the callers allocation context, or to disable the use of caches for NHRTs and/or threads using scopes.

6.4 Consumption of Immortal Memory

The Immortal Memory Area is necessary as it provides an area in which objects that need to be accessed by all threads can be allocated, however, since objects in this area are never collected it also acts as an engineered memory leak. Further, the RTSJ specifies that static initialization is carried out with Immortal as the allocation context and that certain objects such as class objects are always allocated from Immortal. This ensures that Immortal is consumed even if only regular Java code is executed. We found that it was important to carefully consider the functionality within the class libraries such as class loading and the code executed in static initializers so as to minimize the use of Immortal and whenever possible limit it to a one time hit. For example, if a copy of an object in Immortal was required so that it could be accessed by an NHRT it would often be possible to cache this object instead of returning a new copy each time.

7. CLASS LOADING

One of the features of Java is that it loads and initializes classes late, keeping the footprint of applications smaller and ensuring that only what is absolutely required is loaded into memory.

The notion of late binding combined with reflection and JNI makes it very challenging for Real Time Java programmers to ensure that there will be no unexpected delays due to class loading or class initialization along a particular path of code.

There are several ways to address this problem, but in practice, programmers tend to take the brute force approach of manually building a list of classes to preload, based on the classes they think are required for critical sections of their application. This is error prone and unportable, since a programmer may not have access to the source code of a particular library they want to use, and therefore may not be able to determine exactly what classes it actually depends on. Even if they do have access to the source of a particular implementation, when they move to a new version of code, or change to a new vendor, the list of classes required by a critical region could change dramatically.

An easier approach to the problem is to process all the jars, zip files and classes accessible on the command-line and pre-load all of them. This is likely to eliminate class loading from critical sections, but it can take a significant amount of time and memory to process all these classes. Furthermore, there is no easy way to pre-initialize the classes that are loaded since there can be circular dependencies and hidden orderings that are required to initialize the classes correctly.

A better approach is to identify each of the critical regions of the application and determine via static analysis a conservative approximation of all classes that could be referenced on some path through the region. The results of static analysis can be overly conservative because of imprecision introduced by virtual and interface method calls. Our system also depends on help from the programmer to model the affects of reflection and JNI.[2] The analysis detects classes that use reflection or have native methods and require a *proxy* method in these cases. The proxy method mimics the methods and fields that could be reached from the original code. The static analysis code also detects which methods update static data and flag whether it has side effects. For those classes with no side effects, initialization is performed as part of class loading. The handful of classes that have side effects are not pre-initialized, but are grouped separately, in case the programmer wants to manually initialize those few classes, or, preferably, modify the code to remove the side effects.

By using the static analysis approach, either as part of deploying the application or at startup of the application, the programmer no longer needs to maintain a fragile set of classes to preload and instead can focus on the handful of proxy methods and classes with side effects. We provide this Real Time Class Analysis Tool (RaTCAT) through our alphaworks technology as a pure Java implementation that can be used with any JVM.

8. NATIVE CODE COMPILATION

To enable Java programs to perform at levels comparable to or better than languages like C++ or Ada while still preserving platform neutrality, Java methods are typically compiled to native processor instructions by a dynamic, or Just-In-Time (JIT), compiler.

[2]Programmer help is required because the only conservative assumption is that a native method could load any available class.

JIT compiler technology has matured to the point where Java programs can effectively compete with statically compiled languages. JIT compilation activity, however, has two features that are, in fact, drawbacks in the real-time space: 1) compilations compete with the application for CPU resource at unpredictable times, and 2) JIT compilers typically employ inherently speculative optimizations that rely on assumptions that are true when the compilation occurs but that may be violated by later program actions, requiring corrective actions that affect the performance of the generated code. The WebSphere Real Time product includes two compilation strategies, adapted JIT compilation and Ahead-Of-Time(AOT) compilation, to mitigate or eliminate these nondeterministic performance effects so that real-time Java applications can benefit from native code performance levels without risking task deadlines.

8.1 JIT Compilation in a Real-time System

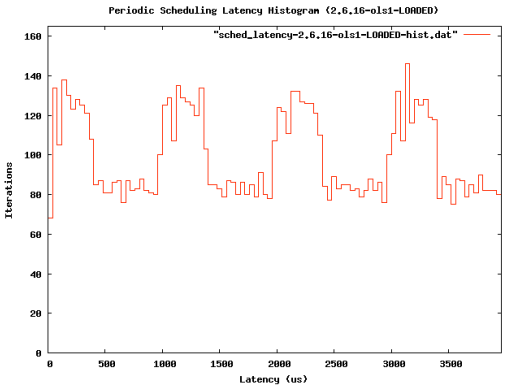
The JIT compiler in the IBM WebSphere Real Time product compiles asynchronously on a separate compilation thread. To prevent it from interfering with real-time tasks, the compilation thread is given the highest *non-real-time* priority. The RT Linux job scheduler therefore guarantees that JIT compilations will only occur whenever there are no higher priority real-time tasks that need to be performed. Context switching times are very low in RT Linux, so this arrangement can be effective for a wide variety of real-time applications. The JIT compiler has also been modified so that no unnecessarily aggressive speculative optimizations are performed. The compiler still performs optimizations like speculative devirtualization to enable method inlining because these optimizations are only invalidated by class load events. Since class loading is dynamic by nature in Java and should be avoided in well designed real-time applications, we felt the tremendous performance benefit from this kind of optimization far outweighed its risk.

8.2 Ahead-of-Time Compilation

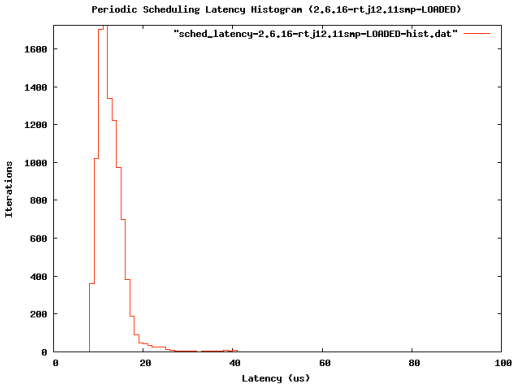
The RT JIT compiler should be suitable for a wide range of softer real-time applications, but there are always situations where even the RT JIT compiler will introduce too much nondeterminism to achieve the goals of a particular application. In these kinds of situations, Ahead-Of-Time compilation is more likely the best approach. The RT AOT compiler can generate native code for methods before the program executes, which means that no JIT compiler is needed at run-time. For this reason, applications employing AOT compilation can achieve even higher levels of deterministic performance. Due to the dynamic nature of the Java language, however, code compiled ahead-of-time may be slower than the code generated by the RT JIT compiler. For example, class references are resolved as the program executes, so the RT AOT compiler cannot rely on the structure of a class at compile-time matching the structure of that class at run-time. A class may have a completely different implementation from one execution to the next, depending on the class path used to invoke the JVM. Alternatively, the class or a superclass may be changed between JVM invocations. For these reasons, the RT AOT compiler cannot use information about fields, methods, or classes while generating code as the RT JIT compiler can. Furthermore, AOT compilation must assume a particular target processor and so the native code performance benefits only apply when the application executes on that particular processor.

9. EVALUATION

IBM Real-time Java is certified for hard real-time performance on a number of AMD-based IBM platforms. We ran all benchmarks on an IBM LS20 blade (model 8850) with two dual-core 2GHz AMD Opteron 270 processors, each with 1 MB of L2 cache.



(a) Stock Linux (2.6.16)



(b) IBM Real-time Linux (2.6.16 RTJ12-SMP)

Figure 2: Thread Scheduling Accuracy

9.1 Operating System Performance

Figure 2 shows the effectiveness of our modifications to the Linux kernel to provide a basis for the real-time virtual machine. The histograms show the accuracy of scheduling a high-priority thread at a 5ms period. 10000 iterations were run on a heavily loaded system building a new Linux kernel. The stock kernel (a) is as much as $4032\mu s$ late, with peaks at millisecond intervals corresponding to the 1ms timer quantum. On the other hand, the real-time kernel (b) is alwyas within $50\mu s$ of the scheduled time.

9.2 Native Code Compilation

To evaluate how well native code compilation benefits application throughput performance for real-time systems, we ran the traditional raw performance benchmarks SPECjvm98 as shown in Figure 3. There are three sets of bars for interpreter-only, JIT native code execution, and AOT native code execution.

In each set, there are two overlapping bars for each benchmark. The low bar for each benchmark shows the fastest time for any execution of the test across 5 runs with 6 iterations in each run. The highest bar shows the slowest time, typically one of the first executions from one of the 5 runs. All times are normalized to the slowest interpreter run for each benchmark. As Figure 3 shows, native code compilation provides a strong performance benefit. JIT compilation provides the best improvement from 2X to almost 26X faster than the interpreter, averaging about a 9.5X speedup. AOT compilation provides a more modest 6.6X performance improvement ranging from 2X to almost 16X. By comparing the difference between the bars for any one benchmark, one can see that the JIT compiler's stronger throughput performance certainly incurs a

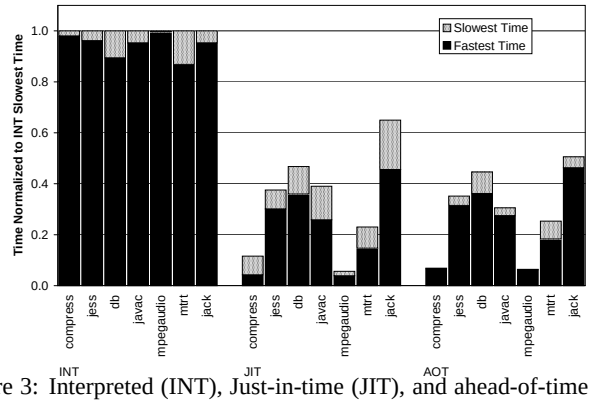


Figure 3: Interpreted (INT), Just-in-time (JIT), and ahead-of-time compiled (AOT) performance for SPECjvm98

Benchmark	Max. Pause	MMU
201.compress	650 μs	72.2%
202.jess	952 μs	69.6%
209.db	770 μs	69.7%
213.javac	738 μs	69.4%
222.mpegaudio	0 μs	100.0%
227.mtrt	868 μs	70.0%
228.jack	706 μs	69.3%
SPECjbb2000	920 μs	67.6%

Table 1: Real-time performance of garbage-collected applications. MMU is for a 10ms time window

penalty in determinism. Some benchmarks, such as compress and jack, improve tremendously because over time the JIT compiler can greatly accelerate the key methods. The AOT generated code, however, provides more steady performance although there are still several benchmarks we believe can be improved (db and mtrt have the most variable performance).

9.3 Real-time Garbage Collection

The Metronome real-time garbage collector makes use of many of the underlying real-time Linux features, including real-time priorities, high-resolution timers, high-resolution clocks, and priority inheritance. Figure 1 shows the worst-case latencies and achieved MMU for standard Java benchmarks. While these are not real-time applications, they are in fact far more allocation intensive than most such applications and thus provide a good stress test of the real-time collector. They also allow comparison with previous work [5, 13].

Worst-case pause times across all eight applications is 920μs (note that the mpegaudio benchmark triggers no collections). This is a thirteen-fold improvement over the original Metronome implementation in Jikes RVM [5], 23 times shorter than those reported for the hardware-assisted Azul system [13], and 163 times shorter than the times reported for BEA's "real-time" JVM [**?**]. Pizlo and Vitek [21] have implemented a real-time collector based on Metronome in the Ovm virtual machine and report worst-case pause times of 1.8ms.

Minimum mutator utilization is never below 67.6% (computed over a 10ms time window, with a target of 70%). By comparison the Jikes RVM Metronome prototype only achieved 45% MMU at 10 ms, and the Azul collector's MMU at 10ms is 0, since it suffers 20ms pauses.

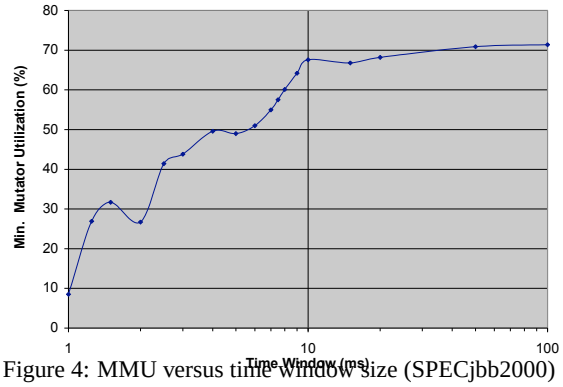


Figure 4: MMU versus time window size (SPECjbb2000)

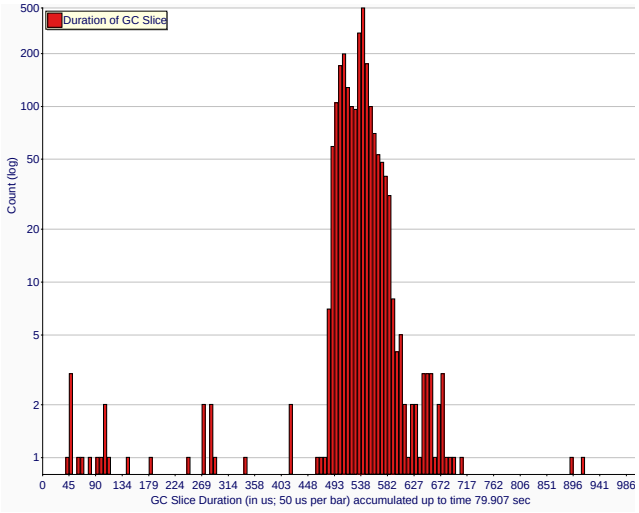


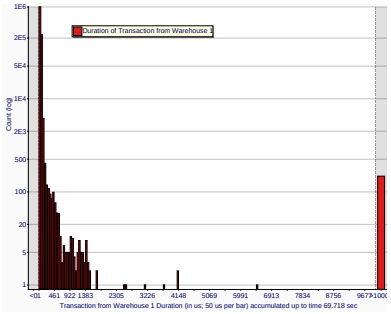
Figure 5: Collector Pause Distribution (SPECjbb2000)

Figure 4 shows the MMU behavior of SPECjbb2000 in detail, and reveals the benefits of over-clocked scheduling of the collector. A usable portion of the CPU (40%) is already available at 2.5ms, reaching 50% at 5ms and 67.6% at 10ms. By contrast, the Jikes RVM-based Metronome system could only achieve 45% MMU at 22.2ms, while the Azul system does not exeeced 50% MMU until it reaches a 200ms time window.

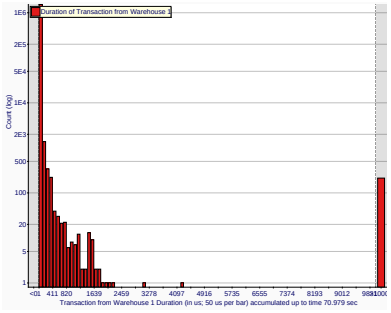
Figure 5 shows the distribution of pause times of collector quanta, which are tightly clustered around the nominal 500μs quantum, but show some jitter due to a combination of inaccuracies in the collector's work estimator, occasional longer atomic quanta, and minor jitter in the underlying operating system scheduling (as evaluated above). However, the pauses never exceed 1ms, and the over-clocked scheduling is able to smooth out the jitter at the quantum-level and provide predictable MMU.

9.4 End-to-end Transaction Time

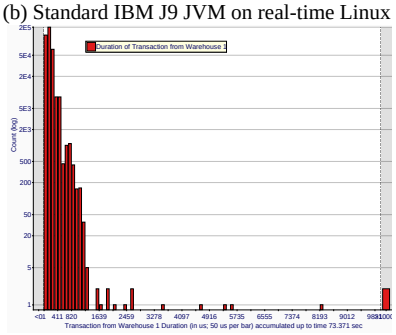
We will now show how the combination of real-time operating system features, real-time garbage collection, thread priorities, and ahead-of-time compilation combine and interact to produce increasingly improved real-time behavior. To evaluate this we use SPECjbb2000 transaction times as an end-to-end metric. While such a transactional system is not representative of traditional periodic real-time control systems like avionics, it is in fact repre-



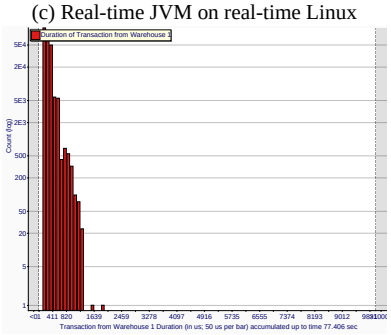
(a) Standard IBM J9 JVM on standard 2.6.16 Linux



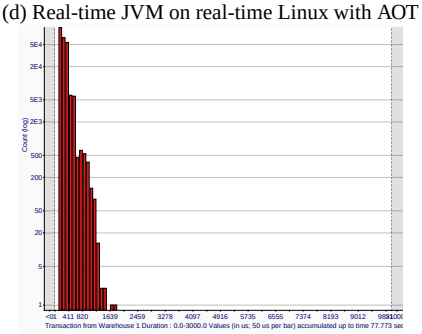
(b) Standard IBM J9 JVM on real-time Linux



(c) Real-time JVM on real-time Linux



(d) Real-time JVM on real-time Linux with AOT



(e) Real-time JVM on real-time Linux with AOT and priorities

Figure 6: Progressive Kernel and JVM Impact on Distribution of SPECjbb2000 Transaction Times

sentative of many of the applications our customers are building, including radar target tracking, financial trading, and SIP (session initiation protocol) processing. We have achieved similar results for more traditional periodic real-time applications such as music synthesis and flight control.

The progression is shown in Figure 6, where the distribution of transaction times is shown up to 10ms, with larger outliers summarized at the right in the grey region. The y-axis is log-scaled. Figure 6(a) shows the distribution of transaction times on the standard IBM J9 JVM running on a stock 2.6.16 Linux kernel. Throughput is high with only a few outliers between 2 and 6 ms, but there are several hundred outliers above 10ms. These are due mostly to garbage collection, but also to JIT compilation. With a standard JVM on top of real-time Linux (b), the distribution up to 10ms is tighter, but the number of very long outliers is not much different.

Beginning with (c) we see the drammatic improvements when using the real-time virtual machine. There are now only 3 outliers above 10ms, and 8 outliers between 2 and 10ms – these are due to JIT compilation, as can be seen in (d), where the application is run with AOT compilation and worst-case transaction time drops to 1.8ms.

Interestingly, all of these measurements are without resorting to using real-time priorities for the application threads. The ability to do so is important because many applications in areas such as financial trading are developed in environments where reliance on the RTSJ extensions is not acceptable to users, who only wish to rely on standard Java APIs. However, we have extended our virtual machine with flags that allow the user to control the default priority for standard java.lang.Threads. Doing so further reduces the worst-case transaction time to about 1.6ms. Obviously, in heavily loaded environments this improvement will be more drammatic.

10. RELATED WORK

We have already compared our system quantitatively to existing Java virtual machines which provide some level of real-time behavior in the previous section. Azul [13] and BEA [**?**] are production systems that implement standard Java but not RTSJ, and provide timing 1-2 orders of magnitude worse than our system. Ovm [21] combines real-time collection and RTSJ into a single virtual machine, but is a research prototype and does not implement the complete Java language and run-time environment.

Other previous work has typically involved implementation of various components of a real-time virtual machine in isolation.

Work on real-time garbage collection began with Baker [6] and was followed by a a variety of implementations with differing levels of timeliness, among them [20, 15, 3, 22, 26, 18]. Cheng [10] introduced the notion of minimum mutator utilization and built a multiprocessor collector with high observed MMU. The Metronome collector [5] upon which our system is based provided the first guaranteed MMU in concert with a strict bound on memory consumption.

Meanwhile work on implementing RTSJ [8] has focused on how to avoid high overheads for special checks required for its memory region checks [4, 7] or avoiding the cost and unreliability through various kinds of static type systems and analysis [2, 14, 9]. The difficulties of using Scopes has also led to investigation of alternative programming models capable of pre-empting the garbage collector [23, 24].

11. CONCLUSIONS

We have presented the IBM Real-time Java virtual machine, which is the first system to combine all of the features required for the construction of large-scale real-time systems in Java: real-time

garbage collection, ahead-of-time compilation, and RTSJ support, and to do so in the kind of environment that the new generation of large-scale real-time systems require: full J2SE environments running on multiprocessors with multiple JVMs per host.

The resulting system achieves excellent real-time behavior and very tight bounds, which has led to its adoption for highly demanding complex real-time systems across a variety of industries.

12. REFERENCES

[1] ALPERN, B., ET AL. The Jalapeño virtual machine. *IBM Syst. J. 39*, 1 (Feb. 2000), 211–238.

[2] ANDREAE, C., COADY, Y., GIBBS, C., NOBLE, J., VITEK, J., AND ZHAO, T. Scoped Types and Aspects for Real-Time Java. In *Proceedings of the European Conference on Object-Oriented Programming* (Nantes, France, July 2006), Springer-Verlag, pp. 124–147.

[3] APPEL, A. W., ELLIS, J. R., AND LI, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices, 23*, 7 (July), 11–20.

[4] ARMBUSTER, A., ET AL. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems* (2006).

[5] 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.

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

[7] BEEBEE, W. S., AND RINARD, M. C. An implementation of scoped memory for real-time Java. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software* (2001), pp. 289–305.

[8] BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[9] BOYAPATI, C., SALCIANU, A., WILLIAM BEEBEE, J., AND RINARD, M. Ownership types for safe region-based memory management in real-time java. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2003), pp. 324–337.

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

[11] 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.

[12] CHENG, P., HARPER, R., AND LEE, P. Generational stack collection and profile-driven pretenuring. In *Proc. of the Conference on Programming Language Design and Implementation* (June 1998). *SIGPLAN Notices, 33*, 6, 162–173.

[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* (Chicago, IL, USA, 2005), pp. 46–56.

[14] DETERS, M., AND CYTRON, R. K. Automated discovery of scoped memory regions for real-time Java. In *Proc. of the Third International Symposium on Memory Management* (Berlin, Germany, June 2002). *SIGPLAN Notices, 38*, 2 Supplement, 25–35.

[15] 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.

[16] FULTON, M., AND STOODLEY, M. Compilation techniques for real-time Java programs. In *CGO 2007: Proceedings of the 2007 International Symposium on Code Generation and Optimization* (2007).

[17] GLEIXNER, T., AND NIEHAUS, D. HRTimers and beyond: Transforming the Linux time subsystems. In *Proc. of the Linux Symposium* (Ottawa, Ontario, June 2006).

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

[19] IBM LINUX TECHNOLOGY CENTER. Real-time linux patches. ftp://linuxpatch.ncsa.uiuc.edu/rt-linux/rhel4u2/R1/rtlinux-src-2006-08-30-r541.tar.bz2.

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

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

[22] 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.

[23] SPOONHOWER, D., AUERBACH, J., BACON, D. F., CHENG, P., AND GROVE, D. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, 2006), pp. 283–294.

[24] SPRING, J. H., PIZLO, F., GUERRAOUI, R., AND VITEK, J. Programming abstractions for highly responsive systems. In *Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (San Diego, California, 2007).

[25] STULTZ, J., HART, D., AND ARAVAMUDAN, N. We are not getting any younger: A new approach to time and timers. In *Proc. of the Linux Symposium* (Ottawa, Ontario, June 2005).

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