Real-time Java, Part 1: Using Java code to program real-time systems

Mark Stoodley
Mike Fulton
Michael Dawson
Ryan Sciampacone
John Kacur

April 10, 2007

This article, the first in a six-part series on real-time Java™, describes the key challenges to using the Java language to develop systems that meet real-time performance requirements. It presents a broad overview of what real-time application development means and how runtime systems must be engineered to meet the requirements of real-time applications. The authors introduce an implementation that addresses real-time Java challenges through a combination of standards-based technologies.

Use of the Java language in real-time systems isn't widespread for a number of significant reasons. These include the nondeterministic performance effects inherent in the Java language's design, such as dynamic class loading, and in the Java Runtime Environment (JRE) itself, such as the garbage collector and native code compilation. The Real-time Specification for Java (RTSJ) is an open specification that augments the Java language to open the door more widely to using the language to build real-time systems (see Related topics). Implementing the RTSJ requires support in the operating system, the JRE, and the Java Class Library (JCL). This article explores the challenges to using the Java language to implement real-time systems and introduces a development kit and runtime environment that tackles those challenges. Subsequent articles in this series will cover in greater depth the concepts and technologies that this article introduces.

Real-time requirements

Real-time (RT) is a broad term used to describe applications that have real-world timing requirements. For example, a sluggish user interface doesn't satisfy an average user's generic RT requirements. This type of application is often described as a soft RT application. The same requirement might be more explicitly phrased as "the application should not take more than 0.1 seconds to respond to a mouse click." If the requirement isn't met, it's a soft failure: the application can continue, and the user, though unhappy, can still use it. In contrast, applications that must strictly meet real-world timing requirements are typically called hard RT applications. An
application controlling the rudder of an airplane, for example, must not be delayed for any reason because the result could be catastrophic. What it means to be an RT application depends in large part on how tolerant the application can be to faults in the form of missed timing requirements.

Another key aspect of RT requirements is response time. It's critical for programmers writing hard or soft RT applications to understand the response-time constraint. The techniques required to meet a hard 1-microsecond response are significantly different from those required to meet a hard 100-millisecond response. In practice, achieving response times below tens of microseconds requires a combination of custom hardware and software, possibly with no -- or a very thin -- operating-system layer.

Finally, designers of robust RT applications typically need some quantifiable level of deterministic performance characteristics in order to architect an application to meet the response-time requirements. Unpredictable performance effects large enough to impact a system's ability to meet an application's response-time requirements make it difficult and maybe even impossible to architect that application properly. The designers of most RT execution environments devote considerable effort to reducing nondeterministic performance effects to meet the response-time needs of the broadest possible spectrum of RT applications.

**Challenges for RT Java applications**

Standard Java applications running on a general-purpose JVM on a general-purpose operating system can only hope to meet soft RT requirements at the level of hundreds of milliseconds. Several fundamental aspects of the language are responsible: thread management, class loading, Just-in-time (JIT) compiler activity, and garbage collection (GC). Some of these issues can be mitigated by application designers, but only with significant work.

**Thread management**

Standard Java provides no guarantees for thread scheduling or thread priorities. An application that must respond to events in a well-defined time has no way to ensure that another low-priority thread won't get scheduled in front of a high-priority thread. To compensate, a programmer would need to partition an application into a set of applications that the operating system can then run at different priorities. This partitioning would increase the overhead of these events and make communication between the events far more challenging.

**Class loading**

A Java-conformant JVM must delay loading a class until it's first referenced by a program. Loading a class can take a variable amount of time depending on the speed of the medium (disk or other) the class is loaded from, the class's size, and the overhead incurred by the class loaders themselves. The delay to load a class can commonly be as high as 10 milliseconds. If tens or hundreds of classes need to be loaded, the loading time itself can cause a significant and possibly unexpected delay. Careful application design can be used to load all classes at application start-up, but this must be done manually because the Java language specification doesn't let the JVM perform this step early.
Stopping the world

Historically, garbage collections are performed while the application program is halted, a process coined Stop-the-world (STW). During a collection, the live objects are traced by starting from a set of “root” objects (those pointed to by static fields, objects that are currently live in some thread’s stack, and so on) and then sweeping unused memory back onto a free list to be used for later allocation requests.

With a STW garbage collector, the application program experiences a GC as a pause in program operation. These STW pauses are unbounded in length and are typically quite intrusive, ranging from hundreds of milliseconds to several seconds. The length of a pause depends on the heap size, the amount of live data in the heap, and how aggressively the collector tries to reclaim free memory.

Many modern collectors use techniques, such as concurrent and incremental algorithms, to help reduce these pause times. But even with these techniques, GC pauses can still occur at indeterminate times with unbounded duration.

Garbage collection

The benefits of GC to application development -- including pointer safety, leak avoidance, and freeing developers from needing to write custom memory-management tooling -- are well documented. However, GC is another source of frustration for hard RT programmers using the Java language. Garbage collects occur automatically when the Java heap has been exhausted to the point that an allocation request can't be satisfied. The application itself can also trigger a collection.

On the one hand, GC is a great thing for Java programmers. Errors introduced by the need to manage memory explicitly in languages such as C and C++ are some of the most difficult problems to diagnose. Proving the absence of such errors when an application is deployed is also a fundamental challenge. One of the Java programming model’s major strengths is that the JVM, not the application, performs memory management, which eliminates this burden for the application programmer.

On the other hand, traditional garbage collectors can introduce long delays at times that are virtually impossible for the application programmer to predict. Delays of several hundred milliseconds are not unusual. The only way to solve this problem at the application level is to prevent GC by creating a set of objects that are reused, thereby ensuring that the Java heap memory is never exhausted. In other words, programmers solve this problem by throwing away the benefits of the managed memory by explicitly managing memory themselves. In practice, this approach generally fails because it prevents programmers from using many of the class libraries provided in the JDK and by other class vendors, which likely create many temporary objects that eventually fill up the heap.

Compilation

Compiling Java code to native code introduces a similar problem to class loading. Most modern JVMs initially interpret Java methods and, for only those methods that execute frequently, later compile to native code. Delayed compiling results in fast start-up and reduces the amount of compilation performed during an application’s execution. But performing a task with interpreted code and performing it with compiled code can take significantly different amounts of time. For
a hard RT application, the inability to predict when the compilation will occur introduces too much nondeterminism to make it possible to plan the application's activities effectively. As with class loading, this problem can be mitigated by using the `Compiler` class to compile methods programmatically at application start-up, but maintaining such a list of methods is tedious and error prone.

**The Real-time Specification for Java**

The RTSJ was created to address some of the limitations of the Java language that prevent its widespread use in RT execution environments. The RTSJ addresses several problematic areas, including scheduling, memory management, threading, synchronization, time, clocks, and asynchronous event handling.

**Scheduling**

RT systems need to control strictly how threads are scheduled and guarantee that they're scheduled deterministically: that is, that threads are scheduled the same way given the same set of conditions. Although the JCL defines the concept of thread priority, a traditional JVM is not required to enforce priorities. Also, non-RT Java implementations typically use a round-robin preemptive scheduling approach with unpredictable scheduling order. With the RTSJ, true priorities and a fixed-priority preemptive scheduler with priority-inheritance support is required for RT threads. This scheduling approach ensures that the highest-priority active thread will always be executing and it continues to execute until it voluntarily releases the CPU or is preempted by a higher-priority thread. Priority inheritance ensures that *priority inversion* is avoided when a higher-priority thread needs a resource held by a lower-priority thread. Priority inversion is a significant problem for RT systems, as we'll describe in more detail in RT Linux®.

**Memory management**

Although some RT systems can tolerate delays resulting from the garbage collector, in many cases these delays are unacceptable. To support tasks that cannot tolerate GC interruptions, the RTSJ defines *immortal* and *scoped* memory areas to supplement the standard Java heap. These areas allow tasks to use memory without being required to block if the garbage collector needs to free memory in the heap. Objects allocated in the immortal memory area are accessible to all threads and are never collected. Because it is never collected, immortal memory is a limited resource that must be used carefully. Scope memory areas can be created and destroyed under programmer control. Each scope memory area is allocated with a maximum size and can be used for object allocation. To ensure the integrity of references between objects, the RTSJ defines rules that govern how objects in one memory area (heap, immortal, or scope) can refer to objects in other memory areas. More rules define when the objects in a scope memory are finalized and when the memory area can be reused. Because of these complexities, the recommended use of immortal and scoped memory is limited to components that cannot tolerate GC pauses.

**Threads**

The RTSJ adds support for two new thread classes that provide the basis for executing tasks with RT behaviour: `RealtimeThread` and `NoHeapRealtimeThread` (NHRT). These classes provide support for priorities, periodic behaviour, deadlines with handlers that can be triggered when the
deadline is exceeded, and the use of memory areas other than the heap. NHRTs cannot access the heap and so, unlike other types of threads, NHRTs are mostly not interrupted or preempted by GC. RT systems typically use NHRTs with high priorities for tasks with the tightest latency requirements, RealtimeThreads for tasks with latency requirements that can be accommodated by a garbage collector, and regular Java threads for everything else. Because NHRTs cannot access the heap, using these threads requires a high degree of care. For example, even the use of container classes from the standard JCL must be carefully managed so that the container class doesn't unintentionally create temporary or internal objects on the heap.

**Synchronization**

Synchronization must be carefully managed within a RT system to prevent high-priority threads from waiting for lower-priority threads. The RTSJ includes priority-inheritance support to manage synchronization when it occurs, and it provides the ability for threads to communicate without synchronization via wait-free read and write queues.

**Time and clocks**

RT systems need higher-resolution clocks than those provided by standard Java code. The new HighResolutionTime and Clock classes encapsulate these time services.

**Asynchronous event handling**

RT systems often manage and respond to asynchronous events. The RTSJ includes support for handling asynchronous events triggered by a number of sources including timers, operating-system signals, missed deadlines, and other application-defined events.

**IBM WebSphere Real Time**

Unlike WebSphere Application Server, WebSphere Real Time does not include a Java Enterprise Edition application server.

Implementing the RTSJ requires broad support from the underlying operating system as well as components of the JRE. IBM® WebSphere® Real Time, released in August 2006 (see Related topics), includes full RTSJ compliance as well as several new technologies aimed at improving RT systems' runtime behaviour and facilitating the work application designers must do to create RT systems. Figure 1 shows a simplified representation of WebSphere Real Time's components:
WebSphere Real Time is based upon IBM's cross-platform J9 technology. Open source RT patches applied to the Linux operating system provide the fundamental RT services required to support RT behaviours, particularly those mandated by the RTSJ. Significantly enhanced GC technology supports 1-millisecond pause times. JIT compilation can be used for softer RT scenarios where compilation can occur when no higher priority work needs to be done. A new **Ahead-of-time (AOT)** compilation technology (not shown in Figure 1) has also been introduced to provide harder RT performance in systems where JIT compilation is inappropriate. The following sections introduce each of these technologies; later articles in this series will give more details on how each technology works.

**RT Linux**

WebSphere Real Time runs on a customized, fully open source version of Linux. Several changes were applied to create an environment for RT Java. These changes provide a fully preemptible kernel, threaded interrupt handlers, high-resolution timers, priority inheritance, and robust mutexes.

**Fully preemptible kernel**

RT Java threads are implemented with **fixed priority** scheduling, also known as **static priority** scheduling, with a first-in-first-out scheduling policy. A standard Linux kernel provides soft RT behaviour, and although there's no guaranteed upper bound on how long a higher-priority thread waits to preempt a lower-priority thread, the time can be roughly approximated as tens of microseconds. In RT Linux, almost every kernel activity is made preemptible, thereby reducing the time required for a lower-priority thread to be preempted and allow a higher-priority one to run. Remaining critical sections that cannot be preempted are short and perform deterministically. RT scheduling latencies have been improved by three orders of magnitude and can now be measured roughly in tens of microseconds.
Threaded interrupt handlers for reduced latency

Almost all interrupt handlers are converted to kernel threads that run in process context. Latency is lower and more deterministic because handlers become user-configurable, schedulable entities that can be preempted and prioritized just like any other process.

High-resolution timers

High-resolution time and timers provide increased resolution and accuracy. RT Java uses these features for high-resolution sleep and timed waits. Linux high-resolution timers are implemented with a high-precision, 64-bit data type. Unlike traditional Linux, where time and timers depend on the low-resolution system tick -- which limits the granularity of timer events -- RT Linux uses independently programmable high-resolution timer events that can be made to expire within microseconds of each other.

Priority inheritance

Priority inheritance is a technique for avoiding the classic priority inversion problem. One of the simplest examples of priority inversion, illustrated in the top diagram in Figure 2, involves three threads: one high (H), one medium (M), and one low (L) priority thread. Imagine H and M are initially dormant waiting for events to be triggered and that L is active and holds a lock. If H wakes up to handle an event, it will preempt L and begin to execute. Consider what happens if H blocks on the lock held by L. Because H cannot make progress until L releases the lock, H blocks and L begins executing again. If M is now triggered by an event, M will preempt L and execute for as long as it needs to. This situation is called priority inversion because M can starve H even though H has higher priority than M.

Figure 2. Example of priority inversion and priority inheritance

RT Linux prevents priority inversion through a policy known as priority inheritance (also known as priority lending), illustrated in Figure 2's bottom diagram. When H blocks on the lock held by L, H gives its priority to L, which guarantees that no task of lower priority than H can preempt L before it releases the lock needed by H. As soon as the lock is released, L's priority reverts to its original value so that H can make progress without waiting further on L. The application designer should still strive to avoid situations where a higher-priority thread requires a resource held by a lower-priority thread, but this priority-inheritance mechanism increases robustness so that priority inversion is prevented.
Robust mutexes and rt-mutexes

Linux pthread mutexes are supported by fast user-space mutexes, known as futexes. Futexes optimize the time to obtain an uncontested lock without relying on the kernel; kernel intervention is required only for contested locks. Robust mutexes solve the problem of cleaning up locks properly after an application holding locks crashes. Also, rt-mutexes extend the priority-inheritance protocol to robust mutexes, which allows the RT JVM to rely on priority-inheritance behaviour via the pthread library.

Deterministic garbage collection

Given an RT operating system, such as RT Linux, that provides the basis for RT behaviours, other major pieces of the JVM can be built to also exhibit RT behaviour. GC is one of the larger sources of nondeterministic behaviour in a JVM, but this nondeterminism can be mitigated through careful design and reliance on the features of RT Linux.

The nondeterministic effects of GC pauses wreak havoc on an RT application's ability to complete tasks under specific deadlines (see Garbage collection). Most GC implementations interfere with an RT application's latency goals to the point where only tasks with larger scale and loose timing requirements can afford to rely on GC technology. The RTSJ's solution to this problem is the introduction of programmer-managed memory allocation via immortal and scope memory areas and NHRTs, but this solution can become a huge headache for Java application designers.

WebSphere Real Time lets programmers rely on the RTSJ memory areas if they desire, but this approach is recommended only for tasks with extremely tight latency requirements. For tasks able to tolerate GC pause times on the order of 1 millisecond, IBM has created deterministic GC technology that lets programmers benefit from the ease of programming with automatic memory management and manage tasks with predictable performance.

IBM's deterministic GC technology is based on two simple premises:

- No single GC pause exceeds some maximum upper bound.
- GC will consume no more than some percentage of any given time window by controlling the number of pauses during that window.

Managing GC activities with these two premises in mind dramatically increases the likelihood that an application can achieve its RT goals.

Metronome GC

WebSphere Real Time uses the Metronome GC to achieve deterministic low-pause-time GC behavior in the JVM (see Related topics). The Metronome GC uses a time-based method of scheduling, which interleaves the collector and the application (known in GC parlance as the mutator because, from the garbage collector's point of view, the application acts to change the graph of live objects over time) on a fixed schedule.

The reason for scheduling against time instead of allocation rate is that allocation is often uneven during an application's execution. By completing GC work as a tax against allocation, it's possible to have uneven distribution of GC pauses and as such reduce the level of determinism in the
GC behaviour. By using time-based scheduling, the Metronome GC can achieve consistent, deterministic, bounded pause times. Further, because no language extensions or modifications to existing code are required, regular Java applications can make use of Metronome transparently and benefit from its deterministic characteristics.

Metronome divides time into a series of discrete quanta, approximately 500 microseconds but no more than 1 millisecond in length, that are devoted to either GC work or application work. Although quanta are very short, if several quanta were devoted to GC work, the application could still experience a longer pause time that might jeopardize RT deadlines. To better support RT deadlines, Metronome distributes quanta devoted to GC work so that the application should receive some minimum percentage of time. This percentage is known as utilization, a parameter the user supplies. Over any time interval, the number of quanta devoted to the application should be no fewer than the specified utilization. By default, the utilization is 70%: in any 10-millisecond time window, at least 7 milliseconds will be devoted solely to the application.

The user can set the utilization at program start-up. Figure 3 shows an example of application utilization over a longer time period. Note the periodic dips corresponding to time quanta where the garbage collector is active. Across the entire time window shown in Figure 3, the application utilization remains at or above the specified 70% (0.7).

**Figure 3. Sample utilization graph**

![Sample utilization graph](image)

Figure 4 demonstrates how deterministic GC pause times are with the Metronome technology. Only a small fraction of pauses exceeds 500 microseconds.
To keep individual GC pauses short, Metronome uses write barriers within the heap and associated metastructures to track live and potentially dead objects. Tracing live objects requires a series of GC quanta to determine which objects should be kept alive and which should be reclaimed. Because this tracing work is interleaved with program execution, the GC can lose track of certain objects that the application can "hide" through executing loads and stores.

This hiding of live objects is not necessarily the result of malicious application code. It's more commonly because the application is unaware of the garbage collector's activities. To ensure no objects are missed by the collector, the GC and VM cooperate by tracking the links between objects as they are created and broken via store operations that the application executes. A write barrier executed before the application performs a store operation does this tracking. The write barrier's purpose is simply to record the change to how objects are linked together if this store could cause a live object to become hidden. These write barriers represent both a performance and memory-footprint overhead that balance the need for deterministic behaviour.

The allocation of large objects can be troublesome for many GC strategies. In many cases, the heap is too fragmented to accommodate a single large object, such as an array. Consequently, it must incur a long pause to defragment, or compact, the heap to coalesce many smaller free memory areas into larger free memory areas to satisfy a large allocation request. Metronome uses a new two-level object model for arrays called arraylets. Arraylets break up large arrays into smaller pieces to make large array allocations easier to satisfy without defragmenting the heap. The arraylet object's first level, known as the spine, contains a list of pointers to the array's smaller pieces, known as leaves. Each leaf is the same size, which simplifies the calculation to find any particular element of the array and also makes it easier for the collector to find a suitable free space to allocate each leaf. Breaking arrays up into smaller noncontiguous pieces lets arrays be allocated within the many smaller free areas that typically occur on a heap, without needing to compact.

Unlike traditional STW garbage collector implementations that have the concept of a GC cycle to represent the start and end of a garbage collect, Metronome performs GC as a continuous process.
throughout the application's lifetime. Application utilization is guaranteed over the application's lifetime, with potentially higher utilization than the minimum in situations where not much GC work is needed. Free memory fluctuates upward and downward as the collector finds free memory to return to the application.

**Native code compilation for RT**

Most modern JVMs use a combination of interpretation and compiled code execution. To eliminate interpretation's high performance cost, a JIT compiler selects frequently executed code to be translated directly to the CPU's native instructions. The Java language's dynamic characteristics typically cause this compiler to operate as the program executes rather than as a step that occurs before the program is run (as is the case for languages like C++ or Fortran). The JIT compiler is selective about which code it compiles so that the time it takes to do the compilation is likely to be made up by the improvements to the code's performance. On top of this dynamic compilation behaviour, traditional JIT compilers employ a variety of speculative optimizations that exploit dynamic characteristics of the running program that might be true at one point during one particular program's execution but might not remain true for the duration of execution. Such optimizations can be "undone" if the assumption about this characteristic later becomes false.

In a traditional non-RT environment, compiling code while the program executes works well because the compiler's actions are mostly transparent to the application's performance. In an RT environment, however, the JIT compiler introduces an unpredictable run-time behaviour that wreaks havoc on worst-case execution time analysis. But the performance benefit of compiled code is still important in this environment because it enables more-complex tasks to complete in shorter periods of time.

WebSphere Real Time introduces two solutions to balance these two requirements at different trade-off points. The first solution is to employ a JIT compiler, operating at a low non-RT priority, that has been modified to perform fewer aggressively speculative optimizations. Operation at a non-RT priority lets the operating system guarantee that the compiler will never interfere with the execution of a RT task. Nonetheless, the fact that the code performance will change over time is a nondeterministic effect that makes this solution more appropriate for softer RT environments rather than hard RT environments.

For harder RT environments, WebSphere Real Time introduces AOT compilation for application programs. Java class files stored in JAR files can be precompiled through a simple command line into Java eXEcutable (JXE) files. By specifying these JXE files, rather than the original JAR files, on the application classpath, the application can be invoked so that the AOT-compiled code is executed -- rather than bytecodes being interpreted or native code being compiled by a JIT compiler. In the first WebSphere Real Time release, using AOT code means that no JIT compiler is present, which has two primary advantages: lower memory consumption and no dynamic performance impact from either the JIT compilation thread or the sampling thread that identifies frequently executing code.

Figure 5 shows how Java code executes in WebSphere Real Time when AOT code is being used:
Starting at the upper left of Figure 5, the developer compiles Java source code to class files as in any Java development project. Class files are bundled into JAR files, which are then AOT compiled using the `jxeinajar` tool. This tool can either compile all the methods in all the classes in the JAR files, or it can selectively compile some of the methods based on output generated by a sample JIT-based execution of the program that identifies the most important methods to compile. The `jxeinajar` tool compiles the methods in a JAR file and constructs a JXE file that contains both the contents of the original JAR file and the native code generated by the AOT compiler. The JXE files can be directly substituted for JAR files when the program is executed. If the JVM is invoked with the `-xnojit` option, then the AOT-compiled code in JXE files on the classpath is loaded (according to the rules of the Java language). During program execution, methods loaded from JAR files or uncompiled methods loaded from JXE files are interpreted. Compiled methods loaded from JXEs execute as native code. In Figure 5, the `-Xrealtime` command-line option is also necessary to specify that the RT VM should be invoked. This command-line option is only available in WebSphere Real Time.

**Disadvantages of AOT code**

Although AOT code enables more-deterministic performance, it also has some disadvantages. The JXEs used to store AOT code are generally much larger than the JAR files that hold the class files because native code is generally less dense than the bytecodes stored in class files. Native code execution also requires a variety of supplementary data to describe how the code needs to be bound into a JVM and how to catch exceptions, for example, so that the code can be executed. A second disadvantage is that AOT-compiled code, though faster than interpreted code, can be substantially slower than JIT-compiled code. Finally, the time to transition between an interpreted method and a compiled method, or vice versa, is higher than the time to call an interpreted method from another interpreted method or to call a compiled method from a compiled method. In a JVM with an active JIT compiler, this cost is eventually eliminated by compiling "around the edges" of the compiled code until the number of transitions is too small to impact performance. In a JVM with AOT-compiled code but no JIT compiler, the number of transitions is determined by the set of methods that were compiled into the JXEs. For this reason, we typically recommend AOT compiling the entire application as well as the Java library classes on which the application depends. Expanding the number of compiled methods, as we mentioned above, has a footprint impact although the benefit to performance is usually more critical than the footprint increase.
The reason AOT code is generally slower than JIT code is because of the nature of the Java language itself. The Java language requires that classes be resolved the first time the executing program references them. By compiling before the program executes, the AOT compiler must be conservative about classes, fields, and methods referenced by the code it compiles. AOT-compiled code is often slower than JIT-compiled code because the JIT has the advantage that it is performing compilation after the executing program has resolved many of these references. However, the JIT compiler must also carefully balance the time it takes to compile a program because that time adds to the program's execution time. For this reason, JIT compilers do not compile all code with the same degree of optimization. The AOT compiler does not have this limitation, so it can afford to apply more-aggressive compilation techniques that sometimes yield better performance than JIT-compiled code. Moreover, more methods can be AOT compiled than a JIT compiler might decide to compile, which can also result in better performance with AOT compilation than JIT compilation. Nonetheless, the common case is that AOT-compiled code is slower than JIT-compiled code.

To avoid nondeterministic performance effects, neither the JIT compiler nor the AOT compiler provided in WebSphere Real Time applies the aggressively speculative optimizations generally applied by modern JIT compilers. These optimizations are generally performed because they can produce substantial performance improvements, but they are not appropriate in a RT system. Furthermore, supporting the various aspects of the RTSJ and the Metronome garbage collector introduces some overheads into compiled code that traditional compilers need not perform. For these reasons, code compiled for RT environments is typically slower than the code compiled for non-RT environments.

**Future directions**

More can be done to make an RT Java environment faster, in terms of both predictable performance and raw throughput. We see two key areas of advancement that must occur for the Java language to succeed in the RT application space:

- Provide RT technology to users who want better predictability while running on traditional operating systems.
- Make it much easier to use this technology.

**Toward soft RT**

Many features of WebSphere Real Time are useful to programmers targeting a traditional operating system. Incremental GC and priority-based threads would clearly be useful in many applications, even if hard RT guarantees could not be met and only soft RT performance could be provided. An application server providing predictable performance without unpredictable GC delays, for example, is an attractive idea to many developers. Similarly, enabling applications to run high-priority Java health-monitoring threads with reasonable scheduling targets would simplify Java server development.

**Making RT easier**

Simply bringing the advantages of using the Java language to the process of creating RT systems is a tremendous benefit to developers. But there's always room for improvement, and we are
constantly evaluating new features that could simplify RT programming even further. You can go to our IBM alphaWorks site to try out our expedited real-time threads research technology that lets developers manage extremely high-frequency events with very little tolerance for variance or delay (see Related topics). The tooling achieves highly deterministic behaviour by preloading, preinitializing, and precompiling the code to handle events and then running the code independently of the garbage collector with fewer and less onerous restrictions than the NHRTs in the RTSJ. You'll also find tooling called TuningFork, which traces paths from the operating system through the JVM and into applications, making it easier to perform detailed performance analysis.
Related topics

- **Real-time Java series**: Read the other parts in this series.
- "A real-time garbage collector with low overhead and consistent utilization" (David F. Bacon, Perry Cheng, and V.T. Rajan, *Proceedings of the 30th Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, 2003*): This paper presents a dynamically defragmenting collector that overcomes the limitations of applying GC to hard RT systems.
- **JSR 1: Real-time Specification for Java**: You'll find the RTSJ at the Java Community Process site.
- "IBM WebSphere Real Time V1.0 delivers predictable response times using Java standards": Read the product announcement for Real Time.
- **The Real-time Linux Wiki**: A self-described "Wiki Web for the CONFIG_PREEMPT_RT community, and real-time Linux in general."
- **WebSphere Real Time**: WebSphere Real Time lets applications dependent on a precise response times take advantage of standard Java technology without sacrificing determinism.
- **Real-time Java technology**: Visit the authors' IBM alphaWorks research site to find cutting-edge technologies for real-time Java.
- **Java SE Real-Time**: Sun Microsystems' commercial RTSJ-comformant implementation.
- **Metronome**: Learn more about Metronome, the GC technology incorporated in WebSphere Real Time.

© Copyright IBM Corporation 2007

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)