11100011100000011110000001111111000010101
01010101010101010101010100000111110101010

Kelvin Nilsen

ADDING REAL-TIME CAPABILITIES *to Java*

Through extensive experimentation, developers somehow find the right combination of parameters to maximize cost, performance, and compliance with real-time constraints.

SUN MICROSYSTEMS INITIALLY DEVELOPED JAVA AS A TOOL TO SUPPORT INTERNAL development of small embedded systems. Later, they determined the language was appropriate for development and distribution of Internet applications and released the language to the general public. Java is object oriented with syntax derived from C and C++ [1, 10], however, Java's designers chose not to pursue full compatibility with C and C++ because they preferred to eliminate from these languages certain troublesome features. In particular, Java does not support enumerated constants; pointer arithmetic; traditional functions, structures and unions; multiple inheritance; goto statements; operator overloading; and preprocessor directives.

Instead, Java requires all constant identifiers and functions (methods) to be encapsulated within class declarations. Java provides standardized support for multiple threads (lightweight tasks) and automatic garbage collection of dynamically allocated memory. Further, Java fully specifies the behavior of every operator on every type, unlike C and C++ which leave many behaviors implementation-dependent. These changes were designed to improve software scalability, reduce software development and maintenance costs, and to achieve full portability of Java software. Anecdotal evidence suggests that many former C and C++ programmers have enthusiastically welcomed these language improvements.

One distinguishing characteristic of Java is its execution model. Java programs are first translated into a fully portable standard bytecode representation. The bytecode is then available for execution in any environment that provides support for a Java virtual machine. A Java virtual machine is simply a system of software that understands and executes the standard Java bytecode representation. All major operating systems now support execution of Java programs, including Windows 95, NT, and CE, Solaris, HP-UX, IRIX, AIW, and MacOS. To prevent viruses from being introduced by a foreign Java bytecode program, the Java virtual machine includes a Java bytecode analyzer that verifies the bytecode but does not contain requests that would compromise the local system. By convention, this bytecode analyzer is applied to every Java program before it is executed. Bytecode analysis is combined with optional run-time restrictions on access to the local file system for even greater security.

Initial Java implementations interpret bytecodes, but recently released implementations provide the ability to translate bytecodes to native machine code on the fly. Sun calls this just-in-time (JIT) compilation. Another technique for improving Java performance, known as ahead-of-time translation, is to translate bytecodes to machine code prior to deploying an application. JIT and ahead-of-time translation techniques offer the potential of providing performance that approximates the execution speed of C++.

To be precise, Java is more than a programming language. Java is a commercial product and the Java name is a trademark of Sun Microsystems. At the time of this writing, only licensees of Sun's Java implementation are allowed to use the Java trademark. Thus the Java name implies much more than a particular programming language syntax; it implies a particular vendor's implementation, a particular execution model (relying upon a standard, intermediate bytecode representation), and a particular collection of standard libraries. According to Sun, the reason it is so restrictive regarding use of the Java trademark is to assure developers that their code will run reliably in every environment claiming the Java name.

Even though the design of Java was originally motivated within Sun by the special needs of embedded systems development and Sun is promoting the use of Java for embedded systems programming, Java, as it has been defined and implemented, lacks important capabilities necessary for the reliable development of portable real-time applications. This article summarizes some of these shortcomings and very briefly describes how these shortcomings are being addressed in a real-time variant of Java that is currently being developed and marketed under the PERC product name. For more information on the PERC real-time API, see www.newmonics.com.

According to Sun, the reason it's so restrictive regarding use of the Java trademark is to assure developers that their code will run reliably in every environment claiming the Java name.

What Is Real-Time Programming?

Computer programs that must execute within particular time constraints are said to be real-time programs or applications. Examples of real-time applications include voice synthesis, air traffic control, control of robots, telephone switching, and full-motion multimedia playback. In hard, real-time systems, there is no tolerance for late (or early) actions. In such systems, it would be better to not provide any response at all than to provide a late response. Coordinating vehicular traffic flow with red traffic-light signals is a hard, real-time activity. A car that enters an intersection after the traffic light has turned red is likely to cause a serious accident. In a soft, real-time application, a late result is generally considered to be better than no result at all. Network packet routers are examples of soft, real-time systems. Though it is desirable to forward packets to the appropriate subnetworks under particular time constraints, it is usually better to delay transmission of a packet than to ignore it entirely.

Special implementation techniques are required of programmers who desire to enforce real-time constraints. In particular, programmers must determine the memory and CPU-time requirements of each real-time task independently, and then must analyze the collective workload of all the tasks that comprise the system workload. One common technique for analyzing system workloads is known as rate monotonic analysis. With rate monotonic scheduling, developers assign task priority in order of decreasing execution frequency. Tasks that execute most frequently are assigned the highest priority. Mathematical analysis of the worst-case scenario (in which all tasks are triggered for execution at the same time) demonstrates that all of the tasks will complete their execution prior to their next period of execution as long as the combined workload represents less than 69% of the CPU's total capacity [4]. Other scheduling techniques, such as earliest-deadline-first, are capable of supporting higher CPU utilizations [3]. Regardless of the analysis technique used, it is necessary to know the worst-case execution time and the maximum execution frequency of each task in order to perform the analysis.

Challenges of Real-Time Development

Developing real-time software is notoriously difficult and very costly. To exercise full control over real-time behavior requires extensive machine-dependent analysis. Because the market for real-time software has traditionally been much smaller than the market for non-real-time software, real-time developers do not enjoy access to the same high-level application development environments that are available to developers of more traditional applications. Much of the analysis required to analyze and demonstrate compliance with real-time constraints must therefore be done by hand rather than by automated tools. Even worse, this analysis must be repeated each time the application is ported to a new architecture or to a new release or configuration of the same architecture.

The effort required to demonstrate compliance with real-time constraints is monumental. It delays time to market and makes development of real-time software much more costly than development of traditional non-real-time software. An additional cost of real-time methodologies is they require much more conservative use of time and memory than is typical of more traditional applications. This is because real-time developers generally must configure time and memory for the worst-case requirements of each real-time activity whereas traditional computer systems are generally configured for typical or average-case resource requirements.

Most real-time practitioners consider the use of formal methodologies to be cost-prohibitive. Rather than use "recommended" methodologies, developers make numerous compromises in the fundamental design and engineering of real-time systems. Unfortunately, there is very little theory to describe the behavior of such compromised systems. As a result, implementation of real-time systems is largely performed through a process of "black magic." Through human wizardry and experimentation, developers find the right combination of parameters that minimizes cost while maximizing performance and compliance with real-time constraints. Because these parameters are determined through arbitrary choices rather than systematic analysis, resulting systems are fragile; following even a very small change to an isolated component of the system workload, it is often necessary to determine new configuration parameters based on extensive retesting of the complete system.

Many real-time programmers now do their development in C and C++. However, these languages do not provide mechanisms to allow programmers to describe real-time constraints. Consequently, programmers are required to enforce real-time requirements using combinations of compile-time analysis tools, pre-run-time measurements of application resource requirements, and run-time interaction with non-standard, real-time operating system services. As a result, the real-time semantics of software written in C and C++ is not represented by the source code alone. Rather, the real-time semantics is scattered throughout specification documents, makefile-driven analyses and source-code transformations, and, on rare occasions, carefully documented logs describing the results of testing and experimentation and justifying the selection of particular configuration parameters. A consequence of this separation between functional and real-time semantics is that maintenance of real-time software is especially difficult. Once a system is considered to be working properly, there is great reluctance to add any new functionality.

Note that development of real-time Java programs is especially difficult. The developer of a Java application has no idea how powerful the CPU and how much memory will be available in the Java virtual machine environment in which the application is to run. Even worse, developers have no ability to predict the combination of other real-time activities that will comprise the total system workload in which their application will be required to run. And even if developers knew exactly what other real-time tasks were to be running in the target execution environment, they would probably not have the opportunity to scrutinize their implementations to analyze how they might interact with the application's code.

Blocking Concerns

In real-time systems, application developers take responsibility for ensuring the software executes on schedule. The required analysis includes consideration of both the time required to execute each task's code and the time each task may have to wait in queues for access to shared resources. Analysis of execution time, discussed previously, is a local concern. Analysis of wait times is a global concern. Determining how much time a task might have to wait in a queue depends on how many other tasks may need to access the same shared resource, the relative priorities of those other tasks, the amount of time each of those tasks will need to retain the shared resource before releasing it, and the times at which each of the other tasks requires access to the shared resources. In general, analyzing intertask timing dependencies is very difficult. And this analysis is especially difficult in the highly dynamic Java execution environment.

In summary, real-time developers analyze two quantities to demonstrate compliance with real-time constraints: execution time and blocking time. Performing accurate analyses is quite difficult. Consider the analogy of a someone attempting to predict how much time will be required to drive a car across a small town. The "execution time" is how long it takes to drive from the starting point to the ending point assuming there are no delays along the way. If the car gets a flat tire or experiences mechanical difficulties, the times required to respond to these problems must be included in the worst-case execution time. But the typical execution time ignores these possibilities. The person's "blocking time" would be the maximum amount of time needed to wait for red lights at intersections, for railroad crossings, for traffic jams, and for coordination with emergency vehicles that are granted priority access to the public roadways. As with execution times, the typical blocking time is much shorter than the worst-case blocking time.

Do Java Developers Need Real-Time?

Currently, most of the Java code being developed is intended for distribution on the Internet. Because the Internet itself is overloaded and unpredictable, users of these Java applications have become accustomed to sluggish, bursty performance. However, Java applications are already under development that will require more predictable real-time performance. These include real-time character animation, computer music synthesis, full-motion audio and video playback, and video conferencing.

Sun is promoting the use of Java as a general-purpose programming language. They are marketing a new operating system called JavaOS for use in small embedded systems. These small systems are to run only applications written in Java. In such systems, real-time constraints will need to be enforced in order to support high-quality mouse tracking and human interaction, fax processing, voice recording and playback, and high-performance network connections. Future hand-held computers will need to provide increasing amounts of real-time functionality. They will likely support pen input, voice understanding and synthesis, and global positioning. JavaOS systems may also be employed in more traditional embedded environments including manufacturing automation, telephone switching, security systems, intelligent air conditioner control, and reactive robots. All of these require varying degrees of compliance with real-time constraints.

Do Real-Time Programmers Need Java?

As discussed, real-time development using current state-of-the-art technologies is largely "black magic." It is also tedious and costly. The resulting systems are inflexible (imposing excessive constraints on allowed input data) and fragile (they are difficult to modify and likely to break if the execution environment changes). Further, development of real-time software is inherently nonportable. Programmers are forced to target particular operating systems and particular hardware architectures.

Java offers important high-level benefits to developers of traditional software. It would be desirable for a real-time dialect of Java to improve upon the state of the practice in real-time development.

A proposal for a real-time dialect of Java has already been published on the Internet [5] and this proposal has been reviewed by hundreds of developers of both traditional embedded systems and of more traditional Internet Java applications. So far, enthusiastic interest and support have been expressed for adding real-time capabilities to the Java language.

Where Java Falls Short

In its current form, Java is not appropriate for the development of real-time software. In order to offer real-time developers the promise of a portable real-time execution environment, certain aspects of the Java language specification that Sun currently describes as undefined need to be more rigorously specified. The following is a list of some of the specific shortcomings of current Java implementations:

Garbage Collection. In a real-time application, it is important that programmers are assured that memory will be available for allocation when new objects need to be allocated. It is also important that background garbage collection not impose arbitrarily long delays at unpredictable times. This would interfere with the developer's ability to demonstrate compliance with real-time constraints. Current Java implementations do not address these issues:

- *Conservative scanning*. To distinguish live objects from garbage, most Java implementations use a partially conservative scanning technique in which memory words containing values that represent legal memory addresses are assumed to represent pointers. Since these words may actually hold integer or floating-point values, the use of conservative scanning techniques may cause the garbage collector to accidentally treat dead objects as live objects, resulting in memory leaks.
- *Fragmentation*. Because most Java implementations use partially conservative garbage collection techniques, it is not possible to relocate live objects in order to defragment the memory heap. Over time, the cumulative effects of fragmenta-

tion may make it difficult to allocate the large objects that are necessary to accomplish real work. This is especially troublesome for embedded systems that are expected to operate reliably for weeks at a time.

- *Scheduling of garbage collection.* In Sun's Java implementation, garbage collection is implemented as a low-priority background thread. If all application threads are I/O bound, this approach works well in that garbage collection is performed during times that the CPU would otherwise be idle. However, if any application tasks are CPU-bound, the garbage collector is not scheduled for execution until the system runs out of memory. The first allocation request that cannot be satisfied from the existing free pool triggers a stop-and-wait garbage collection of the entire system, forcing all other tasks in the system to suspend until garbage collection completes.
- *Lack of system information.* By design, the Java run-time environment does not allow applications to determine how much memory they require or how much total memory is available in the execution environment. This makes it difficult to determine whether applications will run reliably.
- *Failure to budget memory.* In Java it is quite common for multiple independent activities to be running concurrently in a particular execution environment, so it is important for the system's run-time support to enforce memory budgets on each application. Otherwise, one application could allocate and hoard memory that would more appropriately belong to another. But the standard Java libraries provide no ability to request or enforce memory budgets.

Task Scheduling. The traditional Java environment does not provide any mechanisms to allow programmers to specify that tasks should execute at particular times. Applications can request to sleep for a specified number of milliseconds before continuing their execution. However, there is no guarantee the task will be suspended no longer than the requested amount of time, and there is no guarantee the task will have the highest priority at the time it is made ready for execution.

There is also no way for a given Java task to determine how many other tasks are running on the system, their relative priorities, and the fraction of CPU time they consume. Thus, there is no way to assure that a particular task will have sufficient CPU-time resources to execute within its real-time constraints.

Task Synchronization. Java uses monitors (identified by the synchronized keyword) to protect critical sections of code from simultaneous access by multiple tasks. Once a particular task has entered into the monitor corresponding to a particular object, no other task can access that object's monitor code until the first task has exited the monitor. Note that, in order to analyze the time required to perform certain actions, a real-time developer must know how long each task might have to wait for entry to monitors. The information required to perform this analysis is not generally available:

- *Number and priority of competing tasks.* In the highly dynamic Java execution environment, the number and priorities of other tasks sharing access to a particular object are difficult to determine, and may vary throughout the execution of a particular task. It is difficult to determine how many other tasks might be ahead of a particular task in the queue awaiting access to a shared object's monitor.
- *Time spent within monitors.* Java imposes no restrictions on the complexity of code contained within a synchronized method. The code may comprise unbounded loops, dynamic method invocations, and nested entry into other monitors. Newly loaded class libraries may include synchronized statements that lock particular objects. Therefore, it is nearly impossible to determine how much time each competing task might spend within a monitor once its access has been granted.
- *Priority inversion.* Since the specification of Java's standard libraries fails to specify precisely the scheduling model, and fails to specify protocols for avoiding priority inversion [9], real-time programmers would not be able to analyze blocking behaviors even if they did have perfect knowledge of the implementations of all the tasks that comprise the combined system workload.

Run-Time Analysis. Since the time and memory requirements of Java programs are not known until the Java bytecodes have been loaded into the environment in which they are to execute, mechanisms must be provided within the execution environment to analyze these resource requirements. For instance, Java's standard libraries fail to provide a protocol whereby newly loaded tasks would be able to determine how much CPU time they require to execute reliably on the host platform; they also lack mechanisms to enable applications to determine how much memory is required to represent particular objects in the local execution environment. More-over, Java's standard libraries provide no mechanism to enable applications to determine the total system memory capacity, or to determine how much CPU-time and memory are required by the other tasks that are concurrently executing on the host machine.

Although not as fundamental as the problems already mentioned, another challenge faced by developers who are attempting to use Java for the implementation of embedded real-time systems is that design trade-offs made in most current Java implementations have been biased by priorities and mindsets that are inconsistent with their needs. The economies of embedded real-time developers differ significantly from those of traditional desktop application developers. For example, current Java implementations are not space-efficient. Many use a 32-bit word to represent a Java byte [2]. And the choice to use partially conservative garbage collection was biased by a working environment in which memory is abundant and high-speed disk drives make virtual memory readily available. Another example of the tension between desktop and embedded real-time developers is the use of dynamic compilation. In order to achieve high performance, Sun suggests selected code segments be translated from Java bytecodes to native machine language. Selecting which segments to translate is based on recent execution history. Routines that prove themselves to be "hot spots" are translated on the fly. The interruptions required to perform translation, which occur at unpredictable times, complicate analysis of task execution times.

In order to offer real-time developers the promise of a portable real-time execution environment, certain aspects of the Java language specification that Sun currently describes as undefined need to be more rigorously specified.

Adding Real-Time Capabilities

A complete description of the proposed real-time extensions is impractical here—what follows is only a high-level overview of the PERC real-time API, the implementation of which is currently in progress. For more complete descriptions, refer to my earlier work [5, 6].

The real-time API includes mechanisms to enable programmers to analyze and measure the times required to execute particular code segments, to analyze the memory required to represent particular objects, and to abstract access to persistent objects represented by flash or battery-backed RAM. Real-time applications are structured as activities, each of which is comprised of one or more real-time tasks. A typical execution environment would have multiple real-time activities executing at any given time. For example, one activity might be displaying a full-motion television-like news feed while another takes responsibility for tracking the user's pen motions and a third maintains a video conference connection. Each real-time activity is accompanied by configure() and negotiate() methods.

Preparatory to execution of a new real-time activity, the activity is "introduced" to the local real-time executive. The real-time executive, in turn, invokes the activity's configure() method. The responsibility of this method is to determine the activity's resource needs in the local execution environment. This consists of measuring each task's CPU-time requirements and computing the combined memory needs of all the tasks comprising the real-time activity. The configure() method returns to the real-time executive a representation of the activity's minimum and desired resource allocations.

Once the configure() method has returned, the real-time executive endeavors to satisfy the activity's resource requests. The real-time executive proposes a resource budget to the real-time activity by invoking the activity's negotiate() method. Since the real-time executive may propose to budget less resources than were requested by the activity, the activity has the option of rejecting the proposed budget. If the proposed budget is rejected, the real-time executive may decide not to allow the new activity to be added to the system workload. Alternatively, the real-time executive may reclaim resources previously allocated to other activities by renegotiating their resource budgets and then propose a revised budget to the new activity by once again invoking its negotiate() method.

Rather than rely entirely on the use of synchronized code segments, for which blocking times are difficult to analyze in the highly dynamic Java execution environment, PERC provides an additional synchronization mechanism known as an atomic statement.[1] The body of an atomic statement is executed either to completion or not at all. To the programmer, an atomic statement resembles the disabling of interrupts. However, the implementation may differ. In particular, a hard, real-time implementation of PERC might verify that sufficient CPU time remains in the current time slice to complete execution of the atomic statement before allowing control to enter into the atomic statement's body. Without this check, the inability to interrupt the atomic statement might push all other tasks in the system off schedule.

PERC requires the body of an atomic statement be execution-time analyzable. The PERC standard defines a subset of Java for which it is possible, through automated on-the-fly analysis, to determine worst-case execution times. For single-processor implementations, the use of an atomic statement for real-time synchronization scales to larger, more complex software systems much more easily than the use of synchronized statements because there is no need to analyze blocking times. If a particular task has been granted CPU time to execute, then it also has access to whatever atomic statements may lie along its execution path.

A second syntax introduced for the purpose of enabling programmers to describe real-time requirements is a timed statement. The control clause of a timed statement represents an upper boundary on the amount of CPU time the body of the timed statement is allowed to execute. If the body of the timed statement is still executing at the end of its allotted time, the body is aborted by raising a timeout exception.

The following code fragment demonstrates examples of both control structure extensions. In this code, the application refines approximation *x* as many times as it can within a 10ms time budget. The variable *i* counts the number of times *x*'s value is refined. The significance of the atomic control structure in this code is to make sure that the body of the timed statement is not aborted between the assignment to *x* and the increment to *i*.

```
x = computeApproximation();
i = 0;
timed (10 ms) {
   for (;;) {
      z = refineApproximation(x);
      atomic {
         x = z;
         i++;
      }
   }
}
```

A thorough discussion of the special techniques required for a real-time implementation of the Java execution environment would fill a large book. Here we summarize the general principles on which the budgeting of time and memory is based.

Rate-Monotonic Scheduling. Rate-monotonic analysis[2] allows developers of real-time systems to determine whether a collection of real-time tasks will execute within deadlines. The general model is to represent the workload of each task in terms of its worst-case execution time and its maximum execution frequency. Priorities are set for each task so that the task with the highest execution frequency has the highest priority. Other priorities are assigned in order of decreasing execution frequency. Let C_i represent the computation time of task *i*. And let T_i represent the minimum period of execution for task *i*. For example, if task *1* is responsible for drawing frame updates at 20 frames per second and each frame update requires 10ms of CPU time, then C_1 is 10ms and T_1 is (1/20) s = 50 ms. Note that this task utilizes 1/5 = 20% of the system's total CPU time. The total utilization U_{total} of a system of *n* real-time tasks is given by:

$$U_{total} = \sum^{n}\ {}_{i=1}\ {}^{T}{}_{i}$$

As derived in Liu [4], the utilization bound *UB(n)* for this collection of *n* real-time tasks is given by:

$$UB(n)=n\ (2^{1/n}-1)$$

For large *n*, *UB(n)* is approximated by ln 2, which is roughly 69%. As long as $U_{total} \leq UB(n)$, each of the tasks will complete execution prior to the next period in which the task is required to execute [4].

This analysis assumes no tasks block awaiting access to shared data monitors. More sophisticated analyses are available to treat those cases [4]. An important consideration in any system that does on-the-fly schedulability analysis is minimization of the

[1]The atomic statement is not to be confused with database transactions. There is no notion of roll-back or checkpointing implied.

[2]Rate-monotonic is sufficient, but not a necessary technique for scheduling and schedulability analysis.

analysis overhead. Note that the arithmetic required to answer the question of whether a particular workload can be successfully scheduled is quite straightforward and is linear in the number of tasks.

Real-Time Garbage Collection. The goals of real-time garbage collection are to ensure memory is available for allocation at the times the application needs to allocate without interfering with any task's compliance with real-time constraints. A real-time garbage collector must work incrementally, dividing its total effort into many small bursts of work. These bursts of work must be scheduled using the real-time scheduler so as to make sure that the real-time garbage collector makes timely forward progress while at the same time making sure the garbage collector's CPU-time utilization does not intrude upon times set aside for execution of application software.

In order to make sure garbage collection makes adequate forward progress, the total effort required to perform complete garbage collection must be understood. Suppose, for example, that the total available memory is M bytes and that complete garbage collection is known to require S seconds of CPU time. Suppose further that the combined memory requirements of the system's real-time activities is U total bytes, and that the combined allocation throughput is V total bytes of allocation per second. Finally, let R represent the fraction of the CPU time that is dedicated to garbage collection. Note that the real time required to complete incremental garbage collection is S/R.

Consider the state of memory immediately following completion of garbage collection. In the worst steady-state case, there are a total of U bytes of live memory and $V\,(S/R)$ bytes of dead memory currently occupying the heap. If we start the next garbage collection pass as soon as the first has completed, an additional $V\,(S/R)$ bytes of memory will be allocated while this garbage collection pass is executing. Thus, the size of the space required to support this workload, M, measured in bytes, must be greater than or equal to $U+2V(S/R)$. Based on the combined total memory requirement and maximum allocation rates described previously, the minimum fraction of CPU time that must be spent in garbage collection is given by:

$$R \geq \frac{2VS}{M\text{-}U}$$

Note that R is proportional to the maximum rate at which memory is allocated multiplied by the total time required to perform a stop-and-wait garbage collection pass. R is inversely proportional to the difference between the total amount of available memory and the maximum amount of live memory.

Space does not permit a more detailed description of how various garbage collection strategies compare in terms of the symbolic parameters M (as a fraction of the total amount of memory set aside for dynamic memory allocation) and S. For more thorough discussions of real-time garbage collection techniques, see [7, 8].

Summary

Current Java implementations do not provide the mechanisms required for reliable execution of real-time applications. Minor additions to the standard Java libraries and small extensions to the language itself make possible the cost-effective implementation of real-time systems using a variant of the Java language. The capabilities to be offered to real-time developers by a real-time variant of Java represent significant improvements over the current state of the practice. C

References

1. Arnold, K. and Gosling, J. *The Java™ Programming Language. The Java™ Series*, ed. L. Friendly. Addison-Wesley, Reading, Mass., 1996.
2. Arnold, K. and Gosling, J. *Native Methods, in The Java™ Programming Language*, L. Friendly, Ed. Addison-Wesley, Reading, Mass., 1996.
3. Cheng, S.-C. and Stankovic, J.A. Scheduling algorithms for hard real-time systems—A brief survey, in *Tutorial on Hard Real-Time Systems*, J.A. Stankovic and K. Ramamritham, Eds., Computer Society Press of IEEE, Washington, D.C., 1987, p. 618.
4. Liu, C.L. and Layland, J.W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (1973), 44–61.
5. Nilsen, K. *Real-Time Java* (v. 1.1). Iowa State University, Ames, Iowa, 1996; www.newmonics.com).
6. Nilsen, K. Issues in the design and implementation of real-time Java. *Java Developer's J. 1*, 1 (1996), 44–57.
7. Nilsen, K. Progress in hardware-assisted real-time garbage collection. In *Lecture Notes in Computer Science 986*. Springer-Verlag, Kinross, Scotland, 1995.
8. Nilsen, K. Reliable real-time garbage collection of C++. *Comput. Syst.* 7, 4 (1994), 467–504.
9. Sha, L., Rajkumar, R. and Lehoczky, J.P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers 39*, 9 (1990), 1175–1185.
10. Sun Microsystems Inc., *The Java Language Environment: A White Paper*. Sun Microsystems, Inc., Mountain View, Calif., 1995.

Kelvin Nilsen (kdn@newmonics.com) is the founder of NewMonics in Ames, IW, a company that focuses on supporting the PERC a real-time variant of the Java language.