Earliest Deadline First Scheduling for Real-Time Java

Embedded System Conference Europe 2000

Class #: 325

Markus Pilz
esmertec ag, Zurich, Switzerland
mpilz@esmertec.com

Introduction

The Real-Time Specification for Java (RTSJ) by the Java Expert Group supports both event-driven scheduling and time-triggered scheduling, as well as combined systems. Time-triggered scheduling, and specifically earliest-deadline-first scheduling, offers many advantages for the development of hard real-time systems: timing constraints need not to be translated into priorities, deadlines can be guaranteed, and admission testing at runtime is possible. Due to these advantages, it is likely that RTSJ compliant virtual machines supporting earliest-deadline-first scheduling will soon be available.

In this paper, we begin by briefly examining the suitability of Java for embedded systems and by introducing the concept of time-triggered scheduling. We then outline the basic principles of earliest-deadline-first (EDF) scheduling and discuss how EDF was implemented in Jbed, a hard real-time capable Java virtual machine with integrated kernel. Finally, we compare EDF scheduling to priority-based scheduling, as defined by the Real-Time Specification for Java and we discuss synchronization between the event-driven world and the time-triggered-world.

Java for Embedded Systems

There are many reasons to use Java as a software platform for implementing embedded systems: First, Java is a simple, modern object-oriented programming language whit a good security model built into it [1]. Second, dynamic class loading is a central part of Java and readily available. This comes in handy in all the networked embedded devices where the desire of loading bug fixes and additional modules into running systems is emerging very quickly. Third, Java is not only a programming language but a software platform. Through the virtual machine [2] and the standard class libraries [3], an abstraction layer is build between the hardware and the operating system. This abstraction layer is the basis for the high portability of Java and central to the "Write Once, Run Everywhere" (WORA) promise. Especially in the highly fragmented world of embedded and real-time systems, the potential benefits of a hardware and operating system independent platform can not be overestimated.

On the other side, it is not immediately clear that the Java platform can meet the embedded world requirements. First, many Java implementations are simply too memory intensive, too slow and too unpredictable for use in embedded systems. Second, Java is a very high-level language and does not provide any mean for interrupt handling or for accessing memory directly. Third, scheduling is largely unspecified in [1]. Basically, only priority based scheduling with ten different priorities is requested. This was done intentionally to facilitate the porting of the virtual machine to platforms where no good multitasking support is available. For example, many virtual machines running under windows do priority based scheduling without preemption.

Nevertheless, it is possible to use Java in embedded systems. Acceptable size and performance values are possible by choosing the right implementation technology for key components of the Java virtual machine like bytecode execution, garbage collection and scheduling. By carefully considering the interaction between this key components, it is possible to build Java virtual machines that can be used in hard real-time environments [4].

Furthermore, the problems mentioned above are addressed by the Real-Time Specification for Java (RTSJ) [5] in various ways. First, a couple of new abstractions are defined to shield the application from the underlaying implementation technology. Second, a mechanism for asynchronous event handling and asynchronous transfer of control is introduced, and a special class allows to create raw memory that can be accessed directly. Third, the RTSJ defines real-time threads which are scheduled according to some exchangeable policy. The default policy is priority-based, preemptive multitasking with at least 28 distinctive priority levels and FIFO within priority. Furthermore, priority inheritance is required as default algorithm to prevent execution eligibility inversion among Java threads that share some serialized resource.

Priority based scheduling was chosen for backward compatibility with standard Java, but also because event-driven, priority based scheduling is the most widely used scheduling scheme for embedded and real-time programming. At the same time, the RTSJ provides hooks to plug-in other schedulers, like time-triggered, earliest deadline first (EDF) schedulers, which are expected to become very important in the future. Jbed, which is commercially available since 1998, does include both an event-driven and a time-triggered scheduler, which we will describe in the remainder of this paper.

Embedded Real-Time Systems and Time-Triggered Scheduling

By definition, an *embedded systems* is one that is built into some device, which needs to control or interact with its environment. Controlling or interacting with the external world inevitably introduces timing constraints into the system. Thus, given an external event, a reaction is required before some deadline. For example: a data packet arrives over a network and needs to be processed before the next packet arrives 1ms later; a signal needs to be sampled every 2 ms with a precision of 10 us; an interrupt is raised and needs to be serviced within 100 us.

The examples above lead to the definition of a real-time system. A real-time system is one in which the system must satisfy explicit response-time constraints. In other words, the correctness of the system depends on the *timely completion*, in addition to the logical correctness, of the system's tasks. Real-time systems are generally classified as either hard real-time or as soft real-time. A hard real-time system is one in which the failure to meet a timeliness constrain constitutes a failure of the system as a whole. A soft real-time system is one in which the failure to meet a timeliness constraint results in degraded operation of the system.

In the design of real-time systems, timing values such as frequency, worst case execution time and deadline are the natural starting point for application design. These timing values may be derived from specifications or from measurements taken from some existing system or prototype. Only when the application needs to be implemented using a priority based scheduler, are these timing values translated into priorities. Time-triggered scheduling offers the possibility to directly use these timing values without the need to translate them into priorities.

Earliest-Deadline-First Scheduling with Admission Testing

Earliest-deadline-first (EDF) scheduling is a type of time-triggered scheduling, well suited for use in hard real-time environments, which requires that the *duration*, the *deadline*, and the *period* for all tasks be known. A task's duration is defined as the time required to execute the task to completion, whereas its deadline is defined as the time by which it must complete execution. A task's period defines the interval between recurring activations of the task. In figure1, a periodic task *p* is depicted, where the respective deadlines *pd* are just prior to the next activation.

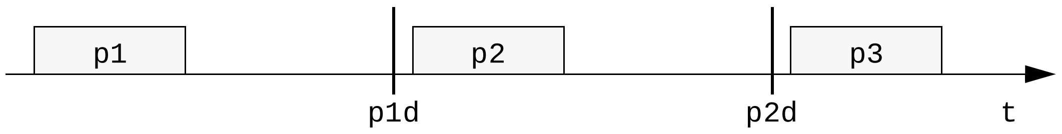

figure 1: A periodic task *p*

Under EDF scheduling, execution eligibility is based on the relative deadlines of all ready tasks, thus the scheduler always runs the task with the closest deadline. In figure 2, a situation is depicted where a new task *q* is installed. The new task *q* preempts *p1*, because its deadline *qd* is closer than *pd1*.

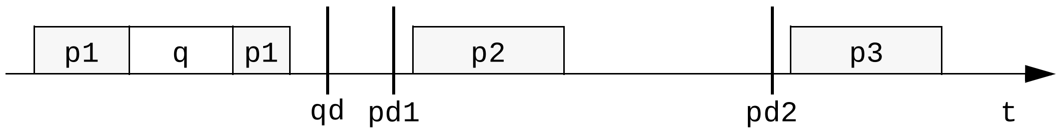

figure 2: The task *q* **preempts** *p*

Given the duration, deadline and period for each task, the EDF scheduler can perform various runtime checks. Most importantly, these values allow the scheduler to perform runtime *admission testing*. Admission testing allows for new tasks to be created and installed dynamically, but only if doing so does not compromise the timeliness constraints of the system. For example, in figure 3, a new task *r* is installed. At the time the new task is created, the scheduler performs an

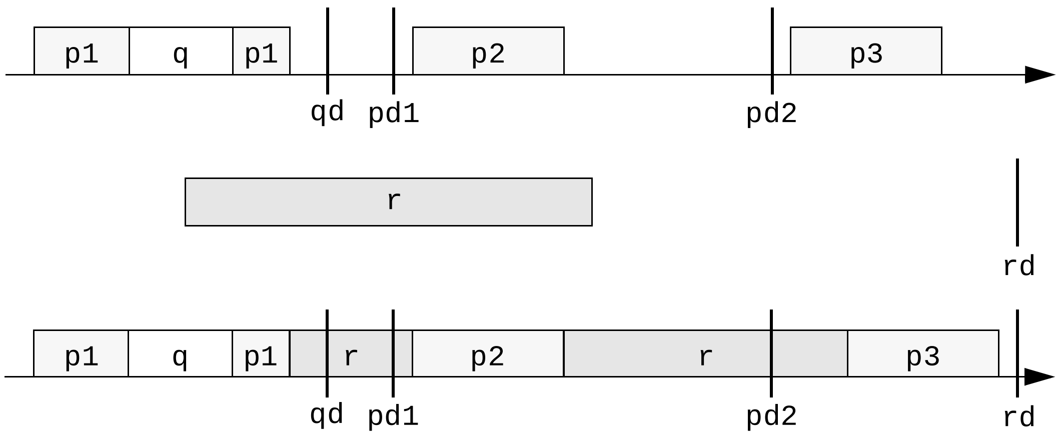

figure 3: Admission testing for r

admission test to check if it can schedule the new task r such that none of the already installed tasks will miss their deadline. In this case, the admission test is passed, but as a consequence, the start of *p3* is delayed.

Admission testing is the key for guaranteeing timeliness constraints in an open, hard real-time system. When admission testing succeeds for a given task, it can be thought of as a contract between the task and the scheduler: they negotiated for the task's execution time (its duration), within a given deadline on a per-period basis. Based on this contract, which is negotiated with all admitted tasks, the scheduler can guarantee the timeliness of all tasks and, thus, of the entire system.

Given that systems generally depend on external inputs (and may suffer from design and/or implementation flaws), it is possible that a particular sequence of events will cause a task to run longer than its negotiated duration or to miss its deadline. In these cases, the scheduler needs to kill the offending task in order to preserve its contract with the other tasks in the system.

Earliest Deadline First Scheduling in Jbed

Jbed is a Java virtual machine with an integrated hard real-time kernel. The Jbed Java virtual machine provides priority- (event-) based scheduling, as specified in the RTSJ, and implements a priority inheritance protocol to avoid priority inversion. In addition, the Jbed hard real-time kernel implements EDF scheduling.

In Jbed, every `Runnable` object executes under the control of either the priority based scheduler or the EDF scheduler. This effectively creates two independent scheduling worlds in the same system. For the remainder of this paper, a *Thread* will refer to a standard Java thread which executes in the event-driven world of priorities while a *Task* will refer to a Jbed task which executes under the control of the EDF scheduler.

A class `com.jbed.tasks.Task` is provided which extends `java.lang.Thread` . All the functionality provided by a Thread is also provided by a Task, but a Task has a different constructor that allows for the specification of duration, deadline, period, and*allowance*. The first three parameters carry the same meaning as in the previous section. The allowance parameter notifies the EDF scheduler to reserve time for executing the exception handler if a `DurationOverflow` or a `DeadlineMiss` exception is thrown.

```
class PDController extends Task {

     public final void run () {
          digitalCoil.write(true); // kick the watchdog
          isPos = - sensor.read();
          isSpeed = (isPos - oldPos) * CLOCK_INVERSE;
          oldPos = isPos;
          outForce = kp * (sollPos - isPos) - kd * isSpeed;
          currentCoil.write(outForce);
     }
}
```

figure 4: Example code for a PD controller

A Task is implemented by extending class `Task` and by overwriting the `run()` method. As a code example, figure4 illustrates part of a simple control application consisting of a PD control task, and figure 5 shows the code which creates and installs the control task.

```
...
try {
     RealtimeEvent event = new PeriodicTimer(2000);

     // duration: 1000, allowance: 0,
     // deadline: 2000, period: event
     controller = new BallController(1000, 0, 2000, event);

     controller.start();

     System.out.println("FloatingBall: controller installed");
} catch (AdmissionFailure e) {
     System.out.pintln(“admission test failed”);
     ...
}
```

figure 5: Installing the controller Task

Typically, the various tasks in the time-triggered world form a network. Jbed does provide various `Task` classes to facilitate the set up of such task networks:

Oneshot. A task that executes only once.

Periodic. A task that executes at regular intervals.

Harmonic. A task that executes as a harmonic of a periodic task.

Join. A task that executes when an other task has completed.

User. A task that executes when a task or thread signals an event.

Interrupt. A task that executes when the hardware signals an event.

The first four task classes function as one would expect. However, interrupt tasks and user tasks both require that a maximum event frequency (i.e., a minimum event period) be specified. The EDF scheduler performs admission testing using the maximum event frequency, even if the actual event frequency stays below the maximum value. The real-time scheduler needs to account for the expected worst case and can not optimize system throughput.

When an interrupt task is triggered, the first action performed is to reset the interrupt, whereas according to its deadline, the interrupt task itself may execute later. Overwriting the reset method in the interrupt task is a convinient way to implement very fast interrupt handlers. Furthermore, interrupt handling outside of the scheduler is provided as alternative to the interrupt task. In this case, an interrupt handler is directly installed and the handler executes outside of any scheduler control, but the maximal load incurred by the interrupt handler needs to be accounted for by the EDF scheduler.

Currently, there is no means for execution eligibility inversion avoidance implemented in the time-triggered world of Jbed. Eligibility inversion avoidance is less important under EDF scheduling, because a task that blocks on a resource locked by a task with a later deadline will receive a deadline miss exception. In general, one solution would be to set the deadline of the

Task that holds the resource to the (closer) deadline of the Task that requests the resource and account the execution time on the duration of the requester. Some kind of ceiling protocol is necessary such that some upper bound for “duration stealing” can be specified.

Synchronization Between the Two Worlds

There is no synchronization possible between the event-driven and the time-triggered world beside some signal send. Every synchronization that makes a Task wait for a Thread breaks the timing guarantee generally available in the time-triggered world. Because the event-driven scheduler does not guarantee any timely completion, a Task waiting for a Thread loses the timeliness guarantee as well and may be aborted by the scheduler through a `DeadlineMiss` exception. In practice, all communication between the event-driven and the time-triggered world needs to be non blocking, for example through the use of a non blocking queue. A Task can signal a Thread through the use of the standard Java `wait()` / `notify()` methods. If `notify()` is called form a Task, a rescheduling takes place and the waiting Thread is signalled imediately. A Thread can communicate with a Task through the `start()` method.

In Java, serializing access to shared resources is done through monitors and the `synchronized` keyword. Locking is done on an object or class basis, depending if the synchronized method (or statement) is an instance or a class method. A thread of execution can enter a synchronized code region only if it holds the corresponding lock. In Jbed, every object is assigned to either the event-driven or the time-triggered world the first time it is locked and depending if the lock is acquired by a Thread or a Task. If the object is first locked by a Thread, it lives in the event-driven, priority based world. If it is first locked by a Task, it lives in the time-triggered EDF world. Based on this assignment, the virtual machine can check for illegal synchronization and signal it to the application by throwing an `IllegalSynchronization` exception at the faulting task.

In the beginning, the restriction to not synchronize between the two worlds may seem restrictive. However, our experience shows, that where there is a big need to synchronize between tasks in their relative world, synchronization between the two worlds is hardly ever required. We arrived at the point where we think that the need to synchronize between the two worlds indicate a design flaw. Synchronization problems indicate that some task is living in the wrong world. Either is there a task without a need for timing guarantees executing in the time-triggered world, or a task that needs to meet some deadline runs in the event-driven world. We would like to let this statement open for further discussion.

Conclusion

Our experience gained by implementing and using Jbed shows that it is easily possible to use Java for programming embedded systems; this includs everything from writing device drivers to implementing hard real-time applications. The prerequisite, however, is a Java implementation which provides sufficient performance, consumes few resources, and provides deterministic behaviour. This is not the case for most Java implementations that are available on the market today.

As of today, Jbed’s real-time features which include EDF scheduling, have been used in over 30 applications like cable assembly robots, power stabilizing equipment, and computer peripherals. These projects have clearly shown the advantages of time-triggered over event-driven scheduling for tasks where timing constraints need to be met. Furthermore, they have shown that admission testing is a key feature for building open systems where additional hard real-time tasks are downloaded and installed on a running and already heavily loaded system. Such open systems are the norm rather then the exception in current embedded Java projects.

In most embedded applications, there are a number of tasks which do not have any hard timing constraints. Instead, it is more natural to rank a task's importance relative to the importance of other tasks. In this case, using a priority- (event-) based scheduler is a perfect fit. Therefore, having a system in which the time-triggered and the event-triggered worlds can coexist is an effective foundation upon which to build an embedded application.

Acknowledgment

Writing this paper was only possible because we had a hard real-time capable Java virtual machine available since long time. I would like to thank the technical staff of esmertec and especially *Daniel Diez* and *Beat Heeb* for having implemented Jbed and for giving me the possibility to write about it. *Mark Stolz* gave me valuable input on an early version of this paper and turned my Swisslish into something very close to English. Many thanks to you all.

Bibliography

[1] J. Gosling, B. Joy, G. Steele. *The Java Language Specification.* Addison Wesley, 1996
[2] T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. 2nd ed., Addison Wesley 1999
[3] D. Flanagan. *Java in an Nutshell*. 2nd ed., O’Reilly, May 1997
[4] J. Tryggvesson, T. Mattsson, H. Heeb. *JBED: Java for Real-Time Systems*. Dr. Dobb’s Journal, Nov. 1999
[5] Real-Time for Java Expert Group. *The Real-Time Specification for Java.* JSR-0001, Addison Weseley, June 2000, available at <http://www.rtj.org>