Real-Time Scheduling on a Java Processor

Martin Schoeberl

JOP.design, Vienna, Austria
martin@jopdesign.com

Abstract. This paper presents the lessons learned by implementing a real-time scheduler for Java on a Java processor. A pure Java system, without an underlying RTOS, is an unusual system with some interesting new properties. Java is a safer execution environment than C (e.g. no pointers) and the boundary between *kernel* and *user space* can become quite loose. Scheduling, usually part of the operating system or the Java Virtual Machine, is implemented in Java and executed in the same context as the application. This property provides an easy path to a framework for user-defined scheduling.

1 Introduction

Java was created as a part of the Green project and specifically intended for an embedded device, a handheld wireless PDA. The device was never released as a product and Java was launched as the new language for the Internet. Over the course of time, Java became very popular for building desktop applications and web applications. However, embedded systems are still programmed in C or C++. The pragmatic approach Java takes to object orientation, its huge standard library and enhancements over C have led to a productivity increase. It now also attracts embedded system programmers. A built-in concurrency model and an elegant language construct to express synchronization between threads also simplify typical programming idioms in this area.

However, some issues remain with Java in an embedded system. Embedded systems are usually too small for Just-In-Time compilation (JIT), resulting in a slow interpreting execution model. Moreover, a major problem for embedded systems that are usually also real-time systems is the under specification of the scheduler. For example, the specification even allows low priority threads to preempt high priority threads. This protects threads from starvation in general purpose applications, but is not acceptable in real-time programming. Even an implementation without preemption is permissible. The aim of this *loose* definition of the scheduler is to be able to implement the Java Virtual Machine (JVM) on a large number of platforms where good multitasking support is not available. The Real Time Specification for Java (RTSJ) [1] addresses many of these problems.

JOP (Java Optimized Processor) is intended to be a solution to the issue of Java's performance in embedded systems. JOP executes Java bytecodes, the instructions of the JVM, in hardware resulting in the efficient execution of Java programs without JIT compilation. It is a tiny processor core, implemented in a Field Programmable Gate Array (FPGA), with architectural features to support real-time systems.

This paper comprises the following elements: Section 2 gives a short overview of JOP that was used as the platform for this work. After the Real-Time Specification for Java is introduced, a subset of this specification for high-integrity real-time systems is defined in Section 3. The results of implementing this specification with JOP are described in Section 4. Drawing general conclusions from this work on scheduling in a Java system leads to a framework for user defined schedulers. Section 5 describes this framework and how to use it. Section 6 sets this paper in context of related work and is followed by a conclusion.

2 A Java Optimized Processor

The Java Optimized Processor (JOP) is the main target for the real-time system described. A detailed description of the architecture can be found at [2]. The processor implements the JVM in hardware. It is intended for embedded real-time systems. JOP has been designed to fit the following constraints:

- Each aspect of the architecture has to be time predictable for Worst Case Execution Time (WCET) analysis and the predictable execution of real-time tasks. Low worst-case execution time is more important than average execution speed.
- The processor has to be small enough to fit into a low cost FPGA device, in order to compete with traditional microcontrollers.

A Java program is compiled to a platform independent representation: a class file containing instructions of the JVM, called bytecodes. These bytecodes are mapped by hardware to JOP microcode instructions. As the JVM instructions set is stack oriented, JOP is implemented as a stack machine. The full-pipelined architecture executes every microcode instruction in a single cycle.

Interrupts are implemented as special bytecodes. These bytecodes are inserted by the hardware in the Java instruction stream. This special bytecode result in a call of a JVM internal method in the context of the interrupted thread. This mechanism implicitly stores almost the complete context of the current active thread on the stack.

The target technology for JOP is an FPGA. With the flexibility of an FPGA, the resource usage and performance of JOP is highly configurable so that it can fit different applications. A typical configuration of JOP uses about 30% of the resources of a low cost FPGA such as Altera's EP1C6 [3]. The remaining resources can be used for application specific peripherals and result in an overall reduction of chip count and board space.

JOP is a pure Java processor, which means it is not necessary to implement native functions in C. All JVM related functions, such as scheduling, are coded in Java. The JVM is self-contained and does not need an operating system. Even a device driver or a network stack is implemented in pure Java.

The implemented JVM is intended to be compatible with the JVM defined in Sun's CLDC (Connection Limited Device Configuration) 1.0 [4]. The library defined in the CLDC is intended for applications in mobile phones. However, traditional embedded systems require a different functionality. Therefore, a different library, coded with conservative memory allocation and thus compatible with the real-time system is provided. For instance, a small TCP/IP stack with its own buffer pooling provides predictable execution time. The predictability is of course at the driver level, not at the transport level.

The processor is fully functioning and used in several real-world applications with different real-time properties. Balfour Beatty Austria has developed a *Kippfahrleitung* [5] to speed up the loading and unloading of goods wagons. JOP is used to control several asynchronous motors to tilt the contact wire up on a line as long as one kilometer. Synchronizing these machines, so that they tilt smoothly, imposes hard real-time constraints on the embedded systems. Another application of JOP is in a communication device with soft real-time properties - Austrian Railways' new security system for single-track lines. Each locomotive is equipped with a GPS receiver and a communication device (that is built with JOP). The position of the train, differential correction data for GPS and commands are exchanged with a server in the central station over a GPRS virtual private network.

3 Java for Embedded Real-Time Systems

The Real-Time Specification for Java (RTSJ) was defined to extend the Java language specification and add predictability. The RTSJ defines new thread classes with a priority based, preemptive scheduler with at least 28 distinct priority levels and FIFO within priorities. To avoid priority inversion, priority inheritance protocol is the mandatory default protocol. Priority ceiling emulation can be used on request for objects to be synchronized.

To avoid the unpredictability of the garbage collector, the RTSJ has introduced new memory models, such as immortal memory and scoped memory. This memory can be used by threads that will not be interrupted by the garbage collector. A new memory area for raw memory access has been introduced to communicate with memory-mapped devices.

Any implementation of the RTSJ must provide the defined priority scheduler and may provide additional schedulers with different policies. User-level schedulers are not part of the RTSJ. An implementation therefore cannot define RTSJ conformant standard classes to provide user-level schedulers.

The real-time system under discussion was inspired by a restricted versions of the RTSJ described in [6] and [7]. It is intended for high-integrity real-time applications and as a test case to evaluate the architecture of JOP as a Java processor for real-time systems. Concurrency is expressed in two types of *schedulable objects*:

Periodic activities are represented by threads that execute in an infinite loop calling waitForNextPeriod() to be rescheduled at predefined time intervals.

Asynchronous sporadic activities are represented by event handlers. Each event handler is, in fact, a thread which is released by an hardware interrupt or a software generated event (invocation of fire()). Minimum interarrival time has to be specified at the creation of the event handler.

All hardware interrupts are represented by a thread under the control of the scheduler. With this solution, a priority is assigned to the device drivers and the execution time can be incorporated in the schedulability analysis with normal tasks. This solution also avoids problems with preemption latency provoked by device drivers. One example of this problem is the *caps-lock* issue in Linux [8]: A device driver performs a spinlock wait for keyboard acknowledgement and produces preemption latency up to 9166 us. With the proposed concept of hardware interrupts under scheduler control, a lower assigned priority to such a device driver avoids preemption delays of *more important* real-time threads and events.

The application is divided in two different phases: *initialization* and *mission*. All non time-critical initialization, global object allocations, thread creation and startup are performed in the initialization phase. All classes need to be loaded and initialized in this phase. The mission phase starts after the invocation of startMission(). The number of threads is fixed and the assigned priorities remain unchanged.

The scheduler is a preemptive, priority-based scheduler with unlimited priority levels and a unique priority value for each schedulable object. The design decision to use unique priority levels, instead of FIFO within priorities, is based on following factors: two common ways to assign priorities are rate monotonic and, in a more general form, deadline monotonic assignment. When two tasks are given the same priority, we can choose one of them and assign a higher priority to that task and the task set will still be schedulable. This results in a strictly monotonic priority order, removing the need to deal with FIFO order. This eliminates queues for each priority level and results in a single, priority-ordered task list with unlimited priority levels.

Synchronized blocks are executed with priority ceiling emulation protocol. Top priority is assumed for an object used for synchronization for which the priority is not set. This avoids priority inversions on objects that are not accessible from the application (e.g. objects inside a library).

The profile does not support a garbage collector. All memory should be allocated at the initialization phase. Without a garbage collector, the heap implicitly becomes immortal memory (as defined by the RTSJ). For objects created during the mission phase, a scoped memory is provided. Each scoped memory area is assigned to one RtThread. A scoped memory area cannot be shared between threads. No references are allowed from the heap to scoped memory. Scoped memory is explicitly entered and left using calls from the application logic. Memory areas are cleared both on creation and when leaving the scope (call of exitMemory()), leading to a memory area with constant allocation time, as opposed to memory with linear allocation time (as the memory type LTMemory in the RTSJ) [9].

To verify that this specification is expressive enough for high-integrity real-time applications, Ravenscar-Java (RJ) [7], with the additional necessary RTSJ classes, has been implemented on top of it. However, RJ inherits some of the complexity of the RTSJ. Therefore, the implementation of RJ has a larger memory and runtime overhead than this simple specification.

```
public class RtThread {

    public RtThread(int priority, int period)
    public RtThread(int priority, int period, int offset)
    public RtThread(int priority, int period, Memory mem)
    public RtThread(int priority, int period, int offset, Memory mem)

    public void enterMemory()
    public void exitMemory()

    public void run()
    public boolean waitForNextPeriod()

    public static void startMission()
}

public class HwEvent extends RtThread {

    public HwEvent(int priority, int minTime, int number)
    public HwEvent(int priority, int minTime, Memory mem, int number)

    public void handle()
}

public class SwEvent extends RtThread {

    public SwEvent(int priority, int minTime)
    public SwEvent(int priority, int minTime, Memory mem)

    public final void fire()
    public void handle()
}
```

Fig. 1. Schedulable objects

4 Implementation Results

The initial idea was to implement scheduling and dispatching in microcode. However, many Java bytecodes have a one to one mapping to a microcode instruction, resulting in a single cycle execution. The performance gain of an algorithm coded in microcode is therefore negligible. As a result, almost all of the scheduling is implemented in Java. Only a small part of the dispatcher, a memory copy, is implemented in microcode and exposed with a special bytecode.

Experimental results of basic scheduling benchmarks, such as periodic thread jitter, context switch time for threads and asynchronous events, can be found in [10]. In this paper, the implementation of this specification on JOP is compared with the reference implementation of the RTSJ on TimeSys RT-Linux.

4.1 Support for the JVM

To implement system functions, such as scheduling, in Java, access to JVM and processor internal data structures have to be available. However, Java does not allow memory access or access to hardware devices. In JOP, this access is provided by way of additional bytecodes. In the Java environment, these bytecodes are represented as static native methods. The compiled call instruction for these methods (invokestatic) is replaced by these additional bytecodes in the class file. This solution provides a very efficient way to incorporate low-level functions into a pure Java system. The translation can be performed during class loading to avoid non-standard class files.

4.2 Context Switch

The time for a context switch depends on the size of the state of the task. For a stack machine it is not very obvious what belongs to the state of a task. If the stack resides in main memory, only a few registers (e.g. program counter and stack pointer) need to be saved and restored. However, the stack is a frequently accessed memory region of the JVM. The stack can be seen as a data cache and should be placed *near* the execution unit (in this case, *near* means on the chip and not in external memory). This means that the stack is part of the execution context and has to be saved and restored on a context switch.

In JOP, the stack is placed in local (on chip) FPGA memory with single cycle access time. With this configuration, the next question is how much of the stack to place there. Either the complete stack of a thread or only the stack frame of the current method can reside locally. If the complete stack of a thread is stored in local memory, the invocation of methods and returns are fast, but the context is large. For fast context switches, it is preferable to have only a short stack in local memory. This results in less data being transferred to and from main memory, but more memory transfers on method call and return. The local stack can be further divided into small pieces, each holding only one stack frame of one thread. During the context switch, only the stack pointer needs to be saved and restored. The outcome of this is a very fast context switch, although the size of the local memory limits the maximum number of threads.

Since JOP is a soft-core processor, these different solutions can be configured for different application requirements. It is even possible to mix of these policies: some stack slots can be assigned to *important* threads, while the remaining threads share one slot. This stack slot only needs to be exchanged with the main memory when switching *to* a less *important* thread.

4.3 JopVM

To simplify debugging a small tool was developed. A simulator of JOP, now called JopVM, was implemented in C. It is compatible with JOP and only needs minimum support from the underlying hardware. JopVM is an interpreting JVM and simulates exact timer interrupts with a high-resolution counter, such as the Pentium time stamp counter.

The Java application being tested can be preverified and linked with JopVM, resulting in a single executable. Although this is a low performing solution for real-time Java, it is time predictable and does not require an operating system. It can be used on any platform where a C compiler, with 32-bit integer support, is available to experiment with the user-defined scheduler framework described in the next section.

5 User-Defined Scheduler

This novel approach implementing a real-time scheduler in Java opens up new possibilities. An obvious next step is to extend this system to provide a framework for user-defined scheduling in Java. New applications, such as multimedia streaming, result in *soft* real-time systems that need a more flexible scheduler than the traditional fixed priority based ones. This work provides a simple to use framework to evaluate new scheduling concepts for these applications in real-time Java.

The following section analyzes which events are exposed to the scheduler and which functions from the JVM need to be available in the user space. It is followed by the definition of the framework and examples of how to implement a scheduler using this framework.

5.1 Schedule Events

The most important element of the user-defined scheduler is to define which events result in the scheduling of a new task. When such an event occurs, the user-defined scheduler is invoked. It can update its task list and decide which task is dispatched.

Timer interrupt: For timed scheduling decisions, a programmable timer generates exact timed interrupts. The time interval for the next interrupt is controlled by the scheduler.

HW interrupt: Each hardware-generated interrupt can be associated with an asynchronous event. This allows the execution of a device driver under the control of the scheduler. Latencies of the device driver can be controlled by assigning the right priority in a priority scheduler.

Monitor: To allow different implementations of priority inversion protocols, hooks for monitorenter and monitorexit are provided.

Thread block: Each thread can cease execution via a call of the scheduler. This function is used to implement methods such as waitForNextPeriod() or sleep(). The reason for blocking (e.g. end of periodic work) has to be communicated to the scheduler (e.g. next time to be unblocked for a periodic task).

SW event: Invoking fire() on an event provides support for signaling. wait, notify of notifyAll are not necessary. However, this mechanism is not part of the scheduling framework. It can be implemented with the user-defined scheduler and an associated thread class.

5.2 Data Structures

To implement a scheduler in Java, some JVM internal data structures need to be accessible.

Object: In Java, any object (including an object from the class Class for static methods) can be used for synchronization. Different priority inversion protocols require different data structures to be associated with an object. Each object provides a field, accessed through a Scheduler method, in which these data structures can be attached.

Thread: A list of all threads is provided to the scheduler. The scheduler is also notified when a new thread object is created or a thread terminates. The scheduler controls the start of threads.

5.3 Services for the Scheduler

The real-time JVM and the hardware platform have to provide some minimum services. These services are exposed through Scheduler:

Dispatch: The current active thread is interrupted and a new thread is placed in the run state.

Time: System time with high resolution (microseconds, if the hardware can provide it) is used for time derived scheduling decisions.

Timer: A programmable timer interrupt (not a timer tick) is necessary for accurate time triggered scheduling.

Interrupts: To protect the data structures of the scheduler all interrupts can be disabled and enabled.

5.4 Class Scheduler

The framework consisting of the class Scheduler has to be extended to implement a user-defined scheduler. The class Task represents *schedulable objects*. For non-trivial scheduling algorithms, Task is also extended. The scheduler lives in normal thread space. There is no special context such as kernel space. The methods of Scheduler are categorized by the caller module and described in detail below.

Application. To use a scheduler in an application, the application only has to create one instance of the scheduler class and has to decide when scheduling starts.

`public` **`Scheduler()`**

A single instance of the scheduler is created by the application.

`public void` **`start()`**

This method initiates the transition to the mission phase of the application. All created tasks are started and scheduled under the control of the user scheduler.

Task. A user-defined scheduler usually needs an associated user-defined thread class (an extension of Task). This class interacts with the scheduler by invoking following methods from Scheduler:

`void addTask(Task t)`

The scheduler has access to the list of created tasks to use at the start of scheduling. For dynamic task creation after the start of the scheduler, this method is called by the constructor of Task, to notify the scheduler to update its list.

`void isDead(Task t)`

The scheduler is notified when a Task returns from the run() method. The scheduler removes this Task from the list of schedulable objects.

`void block()`

Every Task can cease execution via a call of the scheduler. This method is used to implement methods such as waitForNextPeriod() or sleep() in a user defined thread class.

Java Virtual Machine. The methods listed below provide the essential points of communication between the JVM and the scheduler. As a response to an interrupt (hardware or timer), entrance or exit of a synchronized method/block the JVM invokes a method from the scheduler.

`abstract void schedule()`

This is the main entry point for the scheduler. This method has to be overridden to implement the scheduling algorithm. It is called from the JVM on a timed event or a software interrupt (see genInt()) is issued (e.g. when a Task gives up execution).

`void interrupt(int nr)`

The scheduler is notified on a hardware event. It can directly call an associated device driver or use this information to unblock a waiting task.

```
void monitorEnter(Object o)
void monitorExit(Object o)
```

These methods are invoked by the JVM on synchronized methods and blocks (JVM bytecodes monitorenter and monitorexit). They provide hooks for executing dynamic priority changes in the scheduler.

Scheduler. Services of the JVM needed to implement a scheduler are provided through static methods.

`static final void genInt()`

This service from the JVM schedules a software interrupt. As a result, schedule() is called. This method is the standard way of switching control to the scheduler. It is e.g. invoked by block().

```
static final void enableInt()
static final void disableInt()
```

The scheduler cannot use monitors to protect its data structures as the scheduler itself is in charge of handling monitors. To protect the data structures of the scheduler, it can globally enable and disable interrupts.

```
static final void dispatch(Task nextTask, int nextTim)
```

This method dispatches a Task and schedules a timer interrupt at nextTim.

```
static final void attachData(Object obj, Object data)
static final Object getAttachedData(Object obj)
```

The behavior of the priority inversion avoidance protocol is defined by the user scheduler. The root of the Java class hierarchy (java.lang.Object) contains a JVM internal reference of generic type Object that can be used by the scheduler to attach data structures for monitors. The first argument of these methods is the synchronized object.

Scheduler or Task. The following two methods are utility functions useful for the scheduler and the thread implementation.

```
static final int getNow()
```

To support time-triggered scheduling, the system provides access to a high-resolution time or counter. The returned value is the time since startup in microseconds. The exact resolution is implementation-dependent.

```
static final Task getRunningTask()
```

The current running Task (in which context the scheduler is called) is returned by this method.

5.5 Class Task

A basic structure for schedulable objects is shown in Fig. 2. This class is usually extended to provide a thread implementation that fits to the user-defined scheduler. The class Task is intended to be minimal. To avoid inheriting methods that do not fit for some applications, it does not extend java.lang.Thread. However, Task can be used to implement java.lang.Thread.

The methods enterMemory and exitMemory are used by the application to provide scoped memory for temporary allocated objects. Task provides a list of active tasks for the scheduler.

One issue, raised by the implementation of the framework is the way in which access rights to methods need to be defined in Java. All methods, except start(), should be private or protected. However, some methods, such as schedule(), are invoked by a part of the JVM, which is also written in Java but resides in a different package. This results in defining the methods as public and *hoping* that they are not invoked by the application code. The C++ concept of friends would greatly help in sharing information over package boundaries without making this information public.

```
public class Task {

    public Task()
    public Task(Memory mem)
    void start()

    public void enterMemory()
    public void exitMemory()

    public void run()

    static Task getFirstTask()
    static Task getNext()
}
```

Fig. 2. A Basic Schedulable Object

5.6 A Simple Example Scheduler

Fig. 3 shows a full example of using this framework to implement a simple round robin scheduler.

```
public class RoundRobin extends Scheduler {

    //
    //   test threads
    //
    static class Work extends Task {

        private int c;

        Work(int ch) {
            c = ch;
        }

        public void run() {

            for (;;) {
                Dbg.wr(c); // debug output

                // busy wait to simulate
                // 3 ms workload in Work.
                int ts = Scheduler.getNow();
                ts += 3000;
                while (ts-Scheduler.getNow()>0)
                    ;
            }
        }
    }
    //
    //    user scheduler starts here
    //

    public void addTask(Task t) {
        // we do not allow tasks to be
        // added after start().
    }

    //
    //    called by the JVM
    //
    public void schedule() {
        Task t = getRunningTask().getNext();
        if (t==null) t = Task.getFirstTask();
        dispatch(t, getNow()+10000);
    }


    public static void main(String[] args) {

        new Work('a');
        new Work('b');
        new Work('c');

        RoundRobin rr = new RoundRobin();

        rr.start();
    }
}
```

Fig. 3. A very simple scheduler

The only method that needs to be supplied is schedule(). For a more advanced scheduler, it is necessary to provide a combination of a user defined thread class and a scheduler class. These two classes have to be tightly integrated, as the scheduler uses information provided by the thread objects for its scheduling decisions.

5.7 Interaction of Task, Scheduler and the JVM

The framework is used to re-implement the scheduler described in Section 3 and 4. In the original implementation, the interaction between scheduling and threads was simple, as the scheduling was part of the thread class. Using the framework, these functions have to be split to two classes, extending Task and Scheduler. Both classes are placed in the same package to provide simpler information sharing with some protection from the rest of the application. For performance reasons data structures are directly exposed from one class to the other.

The resulting implementation is compatible with the first definition, with the exception that RtThread now extends Task. However, no changes in the application code are necessary.

Fig. 4 is an interaction example of this scheduler within the framework. The interaction diagram shows the message sequences between two application tasks, the scheduler, the JVM and the hardware. The hardware represents interrupt and timer logic. The corresponding code fragments of the application, RtThread and PriorityScheduler are shown in Fig. 5. Task 2 is a periodic task with a higher priority than Task 1.

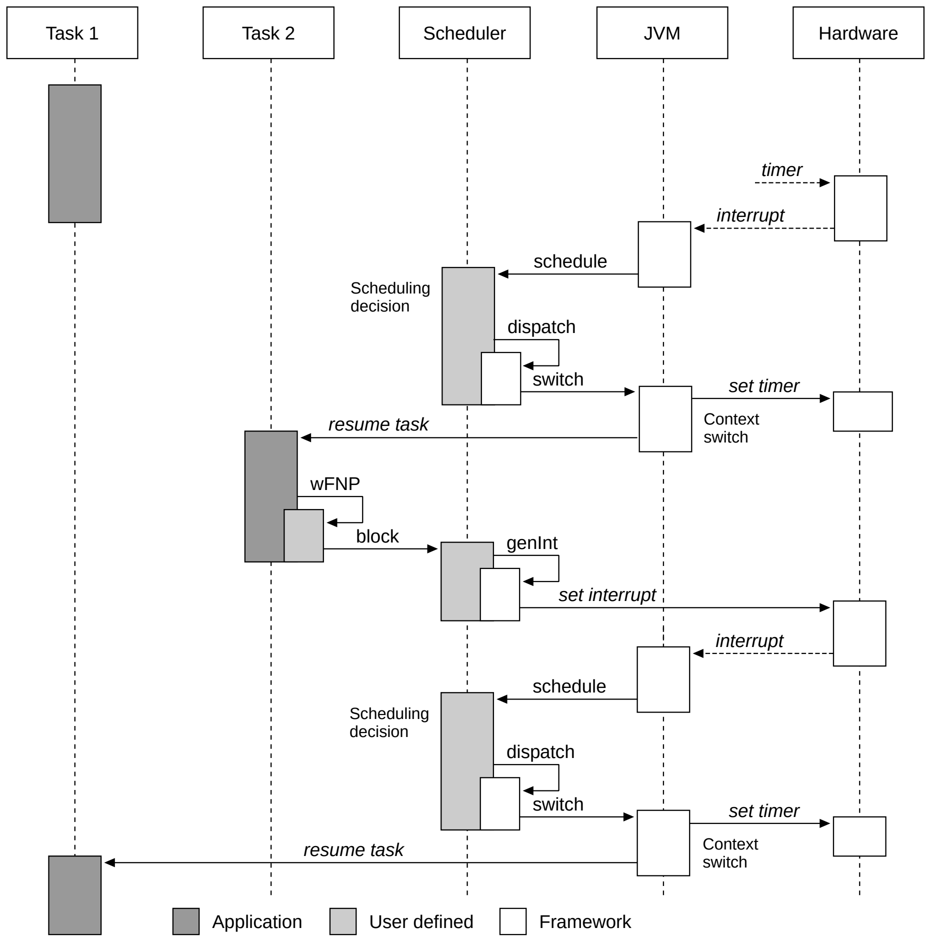

Fig. 4. Interaction and message exchange between the application, the scheduler, the JVM and the hardware.

The first event is a timer event to unblock Task 2 for a new period. The generated timer event results in a call of the user defined scheduler. The scheduler performs its scheduling decision and issues a context switch to Task 2. With every context switch the timer is reprogrammed to generate an interrupt at the next time triggered event for a higher priority task. Task 2 performs the periodic work and ceases execution by invocation of waitForNextPeriod(). The scheduler is called and requests an interrupt from the hardware resulting in the same call sequence as with a timer or other hardware interrupt. The software generated interrupt imposes negligible overhead and results in a single entry point for the scheduler. Task 1 is the only ready task in this example and is resumed by the scheduler.

Using a general scheduling framework for a real-time scheduler is not without its costs. Additional methods are invoked from a scheduling event until the actual dispatch takes place. The context switch is about 20% slower than in the original implementation. It is the opinion of the author that the additional cost is outweighed by the flexibility of the framework.

```
//
//  Application code in the
//  real-time thread:
//

    for (;;) {
        doPeriodicWork();
        waitForNextPeriod();
    }

    ...

//
//  Implementation in RtThread:
//
public boolean waitForNextPeriod() {

    synchronized(monitor) {

        // ps is the instance of
        // PriorityScheduler
        int nxt = ps.next[nr] + period;

        int now = Scheduler.getNow()
        if (nxt-now < 0) {
            // missed deadline
            doMissAction();
            return false;
        } else {
            // time for the next unblock
            ps.next[nr] = nxt;
        }
        // just schedule an interrupt
        // schedule() gets called.
        ps.block();
    }
    return true;
}

    ...

//
//  Implementation in Scheduler:
//
public void block() {
    // Nothing more to do in
    // this implementation.
    genInt();
}

// Generate an interrupt.
// Scheduler gets called from the JVM.
protected static final void genInt() {
    Hardware.interrupt();
}

...
```

```
//
//  Implementation in PriorityScheduler:
//
public void schedule() {

    // Find the ready thread with
    // the highest priority.
    int nr = getReady();

    // Search the list of sleeping threads
    // to find the nearest release time
    // in the future of a higher priority
    // thread than the one that will be
    // released now.
    int time = getNextTimer(nr);

    // This time is used for the next
    // timer interrupt.
    // Perform the context switch.
    dispatch(task[nr], time);
    // No access to locals after this point.
    // We are running in the NEW context!
}
```

Fig. 5. Code fragments from the application, RtThread and the Priority Scheduler

5.8 Predictability

The architecture of JOP is designed to simplify WCET analysis. Every JVM bytecode maps to one ore more microcode instructions. Ever microcode instruction takes exactly one cycle to execute. Thus, the execution time at the bytecode level is known cycle accurate. Most bytecodes have a constant execution time. For some bytecodes, that contain conditional branches in the microcode, the execution time is data dependent. However, the WCET of these bytecode is known. The microcode contains no data dependent or unbound loops that would compromise the WCET analysis.

The worst-case time for dispatching is known cycle accurate on this architecture. Only the time behavior of the user scheduler needs to be analyzed. With the known WCET of every bytecode, the WCET of the scheduler can be obtained by examining it at the bytecode level. This can be done manually or with a tool from XRTJ [11].

6 Related Work

Several implementations of user-level schedulers in standard operating systems have been proposed. In [8], the Linux scheduling mechanism is enhanced. It is divided into a dispatcher and an allocator. The dispatcher remains in kernel space; while the allocator is implemented as a user space function. The allocator transforms four basic scheduling parameters (priority, start time, finish time and budget) into scheduling attributes to be used by the dispatcher. Many existing schedulers can be supported with this parameter set, but others that are based on different parameters cannot be implemented. This solution does not address the implementation of protocols for shared resources.

A different approach defines a new API to enable applications to use application-defined scheduling in a way compatible with the scheduling model defined in POSIX [12]. It is implemented in the MaRTE OS, a minimal real-time kernel that provides the C and Ada language POSIX interface. This interface has been submitted to the Real-Time POSIX Working Group for consideration.

One approach to user-level scheduling in Java can be found in [13]. A thread *multiplexor*, as part of the FLEX ahead-of-time compiler system for Java, is used for utility accrual scheduling. However, the underlying operating system - in this case Linux – can still be seen through the framework and there is no support for Java synchronization.

7 Conclusion

This paper considers the implementation of real-time scheduling on a Java processor. The novelty of the described approach is in implementing functions usually associated with an RTOS in Java. That means that real-time Java is not based on an RTOS, and therefore not restricted to the functionality provided by the RTOS. With JOP, a self-contained real-time system in pure Java becomes possible. This system is augmented with a framework to provide scheduling functions at the application level. The implementation of the specification, described in section 3, is successfully used as the basis for a commercial real-time application in the railway industry. Future work will extend this framework to support multiple schedulers. A useful combination of schedulers would be: one for standard java.lang.Thread (optimized for throughput), one for soft real-time tasks and one for hard real-time tasks.

JOP and the framework for the user-defined scheduler are available at http://www.jopdesign.com. To experiment with this framework, a JOP compatible JVM, written in pure C, is also available. It is intended for debugging and tests with minimum demands on the underlying hardware.

Acknowledgement

I would like to thank Nils Hagge for his port of JopSim, the simulator for JOP, from Java to C. He examined some of the subtle differences between C and Java, which made this translation more difficult than expected.

References

[1] Bollela, Gosling, Brosgol, Dibble, Furr, Hardin and Trunbull. *The Real-Time Specification for Java*, Addison Wesley, 1st edition, 2000.

[2] M. Schoeberl. JOP: a Java Optimized Processor. In *Workshop on Java Technologies for Realtime and Embedded Systems (JTRES 2003)*, Catania, Sicily, Italy, November 2003.

[3] Altera Corporation. *Cyclone FPGA Family*, Data Sheet, ver. 1.2, April 2003.

[4] Sun Microsystems. Java 2 Platform, Micro Edition (J2ME), available at: http://java.sun.com/j2me/docs/

[5] M. Schoeberl. Using a Java Optimized Processor in a Real World Application. In *Proc. Workshop on Intelligent Solutions in Embedded Systems*, Vienna, Austria, June 2003.

[6] P. Puschner and A. J. Wellings. A Profile for High Integrity Real-Time Java Programs. In *Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[7] J. Kwon, A. Wellings and S. King. Ravenscar-Java: a high integrity profile for real-time Java. In *Proc. of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 131-140, Seattle, Washington, USA, 2002

[8] K.J. Lin and Y.C. Wang, The Design and Implementation of Real-Time Schedulers in RED-Linux, In *Proceedings of the IEEE*, Vol. 91, No. 7, July 2003

[9] A. Corsaro, D. Schmidt. The Design and Performance of the jRate Real-Time Java Implementation. Appeared at *the 4th International Symposium on Distributed Objects and Applications*, 2002

[10] M. Schoeberl, Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs. To appear in *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, Gothenburg, Sweden, August 2004.

[11] E. Hu, J. Kwon and A. Wellings. XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment. In *Proc. of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications RTCSA-2003*, pp. 371-391, Tainan, Taiwan, February 2003

[12] M. A. Rivas and M. G. Harbour. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In *Proceedings of 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, IEEE Computer Society Press, pp. 67-75, June 2002

[13] S. Feizabadi, W. BeeBee, B. Ravindran, P.Li and M.Rinard. Utility Accrual Scheduling with Real-Time Java. In *Workshop on Java Technologies for Realtime and Embedded Systems (JTRES 2003)*, Catania, Sicily, Italy, November 2003.