Restrictions of Java for Embedded Real-Time Systems

Martin Schoeberl
JOP.design
martin@jopdesign.com

Abstract

Java, with its pragmatic approach to object orientation and enhancements over C, got very popular for desktop and server application development. The productivity increment of up to 40% compared with C++ [1] attracts also embedded systems programmers. However, standard Java is not practical on these usually small devices. This paper presents the status of restricted Java environments for embedded and real-time systems. For missing definitions, additional profiles are proposed. Results of the implementation on a Java processor show that it is possible to develop applications in pure Java on resource constraint devices.

1. Introduction

Java was first used in an embedded system. In the early '90s Java, which was originally known as Oak, was created as a programming tool for a wireless PDA. The device (known as *7) was a small SPARC based hardware device with a tiny embedded OS. However, *7 was not issued as a product and Java was officially released in 1995 as a new language for the internet (to be integrated into Netscape's browser). Over the years, Java technology has become a programming tool for desktop applications and web services. With every new release, the library (defined as part of the language) continues to grow. Java for embedded systems was clearly out of focus for Sun. With the arrival of mobile phones, Sun again became interested in this embedded market. Sun defined different subsets of Java, which are analyzed in this paper.

As the language became popular, with easier object oriented programming than C++ and threads defined as part of the language, its usage in real-time systems was considered. Two competing groups began to define how to convert Java for use in these systems. Nilsen published the first paper on this subject in November 1995 [2] and formed the Real-Time Working Group. The other group, known as the Real-Time Expert Group, published the RTSJ (Real-Time Specification for Java) [3]. RTSJ was the first specification request under Suns Java Community Process and gained much attention from academic and industrial research. This paper will give:

- An extended overview of actual specifications for Java for embedded and real-time systems
- Propose definitions to fill the gap for small embedded systems, implemented on JOP (a Java Optimized Processor) [4]

This paper is structured as follows. Section 2 summarizes the problems with standard Java on embedded systems. The various definitions for small devices given by Sun are described in Section 3. Section 4 gives an overview of the RTSJ and restrictions for high-integrity applications. If, and how, these specifications are sufficient for small embedded systems in general and specifically for JOP is analyzed. In Section 5 the missing definition for small embedded real-time systems is proposed. Implementation results of this definition and conclusions are presented in Section 6 and 7 respectively.

2. Java Support for Embedded Systems

Java has a built in model for concurrency, the class Thread. All threads share the same heap resulting in a shared memory communication model. Mutual exclusion can be defined on methods or code blocks with the keyword synchronized. Synchronized methods acquire a lock on the object of the method. For synchronized code blocks, the object to be locked is explicitly stated. Every object inherits the methods wait(), notify() and notifyAll() from Object. These methods in conjunction with synchronization on the object support activation. The classes java.util.TimerTask and java.util.Timer (since JDK 1.3) can be used to schedule tasks for future execution in a background thread.

Although Java has language features that simplify concurrent programming the definition of these features is too vague for real-time systems. Java, as described in [5], defines a very loose behavior of threads and scheduling. For example, the specification allows even 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 allowed. Wakeup of a single thread with notify() is not precisely defined. It is not mandatory for a JVM to deal with the priority inversion problem. Garbage collection greatly simplifies programming and helps to avoid classic programming errors (e.g. memo-

ry leaks), but is not suitable for real-time systems and problematic in embedded systems. A more conservative approach to memory allocation is necessary. Dynamic class loading requires the resolution and verification of classes. This function is usually too complex (and too much memory consuming) for embedded devices. Upper bound of execution time for this function is almost impossible to predict (or too large). This results in complete avoidance of dynamic class loading in real-time systems. For an implementation to be Java-conformant, it must include the full library (JDK). The JAR files for this library constitutes about 15 MB (in JDK 1.3, without native libraries), far too large for many embedded systems. Since Java was designed to be a safe language with a safe execution environment, no classes are defined for low-level access to hardware features. The standard library was not defined and coded with real-time applications in mind.

3. Java Micro Edition

The definition of Java also includes the definition of the class library (JDK). This is a huge library and too large for some systems. To compensate for this Sun has defined the *Java 2 Platform, Micro Edition* (J2ME) [6]. As Sun has changed the focus of Java targets several times, the specifications reflect this through their slightly chaotic manner. J2ME reduces the function of the JVM (e.g. no floating point support) to make implementation easier on smaller processors. It also reduces the library (API). J2ME defines three software layers. The *Java Virtual Machine* is assumed to be implemented on top of a host operating system. There are no additional definitions for the J2ME in this layer. The *Configuration* defines the minimum set of JVM features and Java class libraries available on a particular category of devices. In a way, a configuration defines the lowest common denominator of the Java platform features and libraries that the developers can assume to be available on all devices. The *Profile* defines the minimum set of Application Programming Interfaces (APIs) available on a particular family of devices. Profiles are implemented upon a particular configuration. Applications are written for a particular profile and a device can support multiple profiles.

3.1. Connected Limited Device Configuration (CLDC)

CLDC is a configuration for connected devices with at least 192 kB of total memory and a 16-bit or 32-bit processor. As the main target devices are cellular phones, this configuration has become very popular. The CLDC is composed of the K Virtual Machine (KVM) and core class libraries. Error handling has been altered so that the JVM halts in an implementation-specific manner. The following features have been removed from the JVM:

- Floating point support
- Java Native Interface (JNI)
- Reflection
- Finalization
- Weak references
- User-defined class loaders
- Thread groups and daemon threads
- Asynchronous exceptions
- Data type long is optional

These restrictions are defined in the final version 1.0 of the CLDC. The CLDC defines a subset of the following Java class libraries: java.io, java.lang, java.lang.ref and java.util. An additional library (javax.microedition.io) defines a simpler interface for communication than java.io and java.net. When implementing CLDC, one may choose to preload/prelink some classes. A utility (*JavaCodeCompact*) combines one or more Java class files and produces a C files that can be compiled and linked directly with the KVM. There is only one profile defined under CLDC: the Mobile Information Device Profile (MIDP) defines a user interface for LC displays, a media player and a game API.

3.2. Connected Device Configuration (CDC)

The CDC defines a configuration for devices with network connection and assumes a minimum of a 32-bit processor and 2 MB memory. CDC defines no restrictions for the JVM. A virtual machine, the CVM, is part of the distribution. The CVM expects a full featured operating system. The existence of *JavaCodeCompact* implies that preloading of classes is allowed in CDC. Three profiles are defined for CDC. *Foundation Profile* is a set of Java APIs that support resource-constrained devices without a standards-based GUI system. The basic class libraries from the Java standard edition (like java.io, java.lang and java.net) are supported and a connection framework (javax.microedition.io) is added. *Personal Basis Profile* is a set of Java APIs that support resource-constrained devices with a standards-based GUI framework based on lightweight components. It adds some parts of the Abstract Window Toolkit (AWT) support (relative to JDK 1.1 AWT). *Personal Profile* completes the AWT libraries and includes support for the applet interface.

3.3. Discussion

Many of the specifications (i.e. *configurations* and *profiles*) are developed using the Java Community Process (JCP). JCP is not an open standard nor is it part of the open source concept. Although the acronym J2ME implies Java version 2 (i.e. JDK 1.2 and later) almost all technologies under J2ME are still based on JDK 1.1. Besides Java Card, CLDC is the 'smallest' definition from Sun. It assumes an operating system and is quite large (the JAR file

for the classes is about 450 kB). There are no API definitions for low-level hardware access. CLDC is not suitable for small embedded devices. Java Card defines a different JVM instruction set and thus compromises basic ideas of Java. A more restricted definition with following features is needed:

- JVM restrictions, such as in CLDC 1.0
- A package for low-level hardware access
- A minimum subset of core libraries
- Additional profiles for application domains

4. Real-Time Extensions

In 1999, a document defining the requirements for real-time Java was published by NIST [7]. Based on these requirements, two groups defined specifications for real-time Java. A comparison of these two specifications and as compared with Ada 95's Real-Time Annex can be found in [8]. The Real-Time Core Extension [9] is a specification published under the J Consortium. It is still in a draft version and there is no implementation available.

The Real-Time Specification for Java (RTSJ) defines a new API with support from the JVM [3]. A Reference Implementation (RI) of RTSJ forms part of the specification. RTSJ is backward compatible with existing non-real-time Java programs, which implies that RTSJ is intended to run on top of J2SE (and not on J2ME). The following section presents an overview of the RTSJ.

4.1. Threads and Scheduling

A priority-based, preemptive scheduler with at least 28 real-time priorities is defined as base scheduler. Additional levels (ten) for the traditional Java threads need to be available. Threads with the same priority are queued in FITO order. Additional schedulers (e.g. EDF) can be dynamically loaded. The class Scheduler and associated classes provide optional support for feasibility analysis.

Any instances of classes that implement the interface Schedulable are scheduled. In RTSJ RealtimeThread, NoHeapRealtimeThread, and AsyncEventHandler are *schedulable objects*. NoHeapRealtimeThread has and AsyncEventHandler can have a priority higher than the garbage collector. As the available release-parameters indicate, threads are ether periodic or asynchronous events. Threads can be grouped together to bind the execution cost and deadline for a period.

4.2. Memory

As garbage collection is problematic in real-time applications, RTSJ defines new memory areas. *Scoped memory* is a memory area with bound lifetime. When a scope is entered (with a new thread or through enter()), all new objects are allocated in this memory area. Scoped memory areas can be nested and shared among threads. On exit of the last thread from a scope, all finalizers of the allocated objects are invoked and the memory area is freed. *Physical memory* is used to control allocation in memories with different access time. *Raw memory* allows byte-level access to physical memory or memory-mapped I/O. *Immortal memory* is a memory shared between all threads without a garbage collector. All objects created in this memory area have the same live time as the application (a new definition of immortal). *Heap memory* is the traditional garbage collected memory. Maximum memory usage and the maximum allocation rate per thread can be limited. Strict assignment rules between the different memory areas have to be checked by the implementation.

4.3. Synchronization

The implementation of synchronized has to include an algorithm to prevent priority inversion. Priority inheritance is the default and priority ceiling emulation protocol can be used on request. Threads waiting to enter a synchronized block are priority and FIFO within priority ordered. Wait free queues are provided for communication between instances of java.lang.Thread and RealtimeThread.

4.4. Time and Timers

Classes to represent relative and absolute time with nanosecond accuracy are defined. All time parameters are split to a long for milliseconds and an int for nanoseconds within those milliseconds. Each time object has an associated Clock object. Multiple clocks can represent different sources of time and resolution. This allows reduction of queue management overheads for tasks with different tolerance for jitter. A new type, rationale time, can be used to describe periods with a requested resolution over a longer period (i.e. allowing release jitter between the points of the *outer* period). Timer classes can generate time-triggered events (one shot and periodic).

4.5. Asynchrony

Program logic representing external world events is scheduled and dispatched by the scheduler. An AsyncEvent object represents an external event (such as a POSIX signal or a hardware interrupt) or an internal event (through call of fire()). Event handlers are associated to these events and can be bound to a regular real-time thread or represent something *similar* to a thread. The relationship between events and handlers can be many-to-many. Release of handlers can be restricted to a minimum interarrival time.

The exception handling of Java is extended to represent asynchronous transfer of control (ATC). RealtimeThread overloads interrupt() to generate an AsynchronousInterruptedException (AIE). The AIE is deferred until the execution of a method that is willing to accept ATC. The method

indicates this by including AIE in its throw clause. The semantics of catch is changed so that, even when it catches an AIE, the AIE is still propagated until the happened() method of the AIE is invoked. Timed, a subclass of AIE, simplifies the programming of timeouts.

4.6. Discussion of the RTSJ

RTSJ is a complex specification leading to a big memory footprint. The core classes and the JVM executable of the RI on Linux constitute about 5 MB. RTSJ assumes a RTOS and the RI runs on a heavyweight RT-Linux system. RTSJ is too complex for low-end embedded systems. This complexity also hampers programming of high-integrity applications. Runtime memory allocation of the RTSJ classes is not documented.

Threads and Scheduling: If a real-time thread is preempted by a higher priority thread, it is not defined if the preempted thread is placed in front or back of the waiting queue. It is not specified whether the default scheduler performs, or has to perform, time slicing between threads of equal priority.

Memory: It would be ideal if real-time systems were able to allocate all memory at the initialization phase and forbid dynamic memory in the mission phase. However, this is too restrictive for some library functions. The solution to this problem in RTSJ is ScopedMemory, a memory space with limited lifetime. However, it can only be used as a parameter for thread creation or with enter(Runnable r). In a system without dynamic thread creation, using scoped memory at creation time of the thread leads to the same behavior as using immortal memory. The syntax with enter() leads to a cumbersome programming style: for each code part where limited lifetime memory is needed a new class has to be defined and a single instance of this class allocated at initialization time. A simpler syntax is shown in Figure 1. The main drawback of this syntax is that the programmer is responsible for its correct usage.

```
public void run() {
    ...
    myMem.enter();
    {   // new code block disables access
        // to new objects in outer scope.
        Abc a = new Abc();
    }
    myMem.exit();
    ...
}
```

Figure 1: Simpler syntax for scoped memory

Time and Timers: Why is the time split into milliseconds and nanoseconds? In the RI, it is converted to ns for add/subtract. After all mapping and converting (AbsoluteTime, HighResolutionTime, Clock and RealTimeClock) the System.currentTimeMillis() time is used. Since time triggered release of tasks can be modeled with periodic threads, the additional concept of timers is superfluous.

Asynchrony: An unbound AsyncEventHandler is not allowed to enter() a scoped memory. However, it is not clear if scoped memory is allowed as a parameter in the construction of a handler. An unbound AsyncEventHandler leads to the implicit start of a thread on an event. This can lead to substantial overheads. From the application perspective, bound and unbound event handlers behave in the same way. This is an implementation hint expressed through different classes. A consistent way to express the *importance* of events would be a scheduling parameter for the minimum allowed latency of the handler. The syntax that is used in the throws clause of a method to state that ATC will be accepted is misleading. Exceptions in throws clauses are usually generated in a method and not *accepted*.

J2SE Library: It is not specified which classes are safe to be used in RealTimeThread and NoHeapRealTimeThread. Several operating system functions can cause unbound blocking and their usage should be avoided. The memory allocation in standard JDK methods is not documented and the use in immortal memory can lead to memory leaks.

Missing Features: There is no concept such as start mission. Changing scheduling parameters during runtime can lead to inconsistent scheduling behavior. There is no provision for low-level blocking such as disabling interrupts. This is a common technique in device drivers where some hardware operations have to be atomic without affecting the priority level of the requesting thread.

On Small Systems: Many embedded systems are still built with 8 or 16-bit CPUs. 32-bit processors are seldom used. The default integer type of Java is 32-bit, still large enough for almost all data types needed in embedded systems. Why are, often expensive, longs (64 bit integer) used in the RTSJ?

4.7. Subsets of RTSJ

RTSJ is complex to implement and applications developed with RTSJ are difficult to analyze (because of some of the sophisticated features of the RTSJ). Various profiles have been suggested for high-integrity real-time applications that result in restrictions of the RTSJ. In [11], a subset of the RTSJ for high-integrity application domain with hard real-time constraints is proposed. It is inspired by the Ravenscar profile for Ada [12] and focuses is on exact temporal predictability.

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 returning from main(), which is assumed to execute with maximum priority. The number of threads is fixed and the assigned priorities remain unchanged.Two types of tasks are defined: *Periodic time-triggered activities* execute an infinite loop with at least one call of waitForNextPeriod(). *Sporadic activities* are modeled with a new class SporadicEvent. A SporadicEvent is bound to a thread and an external event on creation. Unbound event handlers are not allowed. It is not clear if programmatic trigger of the event is allowed (invocation of fire()). A restriction for a minimum interarrival time of events is not defined. Timers are not supported as time-triggered activities are well supported by periodic threads. Asynchronous transfers of control, overrun and miss handles and calls to sleep() are not allowed. Synchronized methods with priority ceiling protocol provide mutual exclusion to shared resources. Threads are dispatched in FIFO order within each priority level. Sporadic events are used instead of wait, notify and notifyAll for signaling. Since garbage collection is still not time predictable, it is not supported. Scoped memory (LTMemory) is provided for object allocation during the mission phase. LTMemory has to be created during the initialization phase with initial size equal maximum size. For each thread and for the operations of the JVM WCET must be computable. Code is restricted to bound loops and bound recursions. Annotations for WCET analysis are suggested. The JVM needs to check the timing of events and thread execution. It is not stated how the JVM should react to a timing error.

Ravenscar-Java profile [13] is a restricted subset of RTSJ and is based on the work mentioned above. To simplify the initialization phase RJ defines Initializer, a class that has to be extended by the application class which contains main(). The use of time scoped memory is further restricted. LTMemory areas are not allowed to be nested nor shared between threads. Traditional Java threads are disallowed by changing the class java.lang.Thread. The same is true for all schedulable objects from the RTSJ. Two new classes are defined:

- PeriodicThread where run() gets called periodically, removing the loop construct with waitForNextPeriod().
- SporadicEventHandler binds a single thread with a single event. The event can be an interrupt or a software event.

Criticisms of Subsets of the RTSJ: If a subset of RTSJ is implemented and allowed it is harder for programmers to find out what is available and what not. This form of *compatibility* causes confusion. The use of different classes for a different specification is clearer and less error prone. Ravenscar-Java, as a subset of the RTSJ, claims to be compatible with the RTSJ, in the sense that programs written according to the profile are valid RTSJ programs. However, mandatory usages of new classes such as PeriodicThread need an emulation layer to run on a RTSJ system. In this case, it is better to define complete new classes for a subset and provide the mapping to RTSJ. It is not necessary to distinguish between heap and immortal memory. Without a garbage collector, heap implicitly equates to immortal memory.

5. Definitions for Small Embedded Systems

Restrictions on Java, such as the available configurations and profiles of J2ME and the RTSJ, are still too large and complex for small embedded real-time systems. Simpler definitions are therefore proposed: a *configuration* restricts the function of the JVM and the necessary library. Class definitions for low-level I/O are added. On top of this configuration, a *profile* for high-integrity real-time applications is defined.

5.1. Small Embedded Devices Configuration

SEDC is a configuration that fits into the J2ME layer model. It is intended for small embedded devices with a 16-bit (or even 8-bit) microprocessor and a low memory budget (below 128 kB).

Restrictions of the JVM: The JVM restrictions are similar to CLDC 1.0. The subset of the Java classes is smaller. To simplify the JVM the application is preverified and preloaded (and sometimes also linked with the JVM). Since small embedded devices have no or only a very simple user interface (like switches and LEDs), no stream input/output facilities are usually necessary. As memory is very limited, the Java class library is reduced to the absolute minimum. Only the following classes need to be available: java.lang.Object and java.lang.String. Threads are not part of SEDC. We define new thread behavior with real-time semantics in an extra profile. An additional library is provided for low-level I/O-access:

Class IOPort: Static methods of this class allow the application low-level access to I/O ports:

```
public static int read(int address)
public static void write(int value, int address)
```

Class Clock: Almost all microcontrollers have some kind of timer or counter. The class Clock provides a standard way of querying counter values. count() returns the current time in clock ticks. Tick frequency is device dependent and can be queried. ticksPerSecond() returns the resolution of the clock. For a system with a clock frequency higher than 2.15 GHz ticksPerMs() provides the resolution of the clock per millisecond. resolutionInBits() returns the length of the internal clock in bits.

Class Interrupt: The interrupt handler must extend the class Interrupt and register itself for an interrupt. An inter-

rupt causes the object's handle() method to be called. For simple blocking, all interrupts can be disabled. The individual masking of interrupts is device specific and is handled via IOPort.

5.2. A Real-Time Profile for Embedded Java

A simple real-time profile is defined in the concept of the ADA Ravenscar profile [12]. It resembles the ideas from [11] and [13] but is not compatible with the RTSJ. Since the application domain for RTSJ is different from high-integrity systems, it makes sense for it *not* to be compatible with RTSJ. Restrictions can be enforced defining new classes (e.g. setting thread priority in the constructor of a real-time thread alone, enforcing minimum interarrival times for sporadic events). This profile addresses the same devices as SEDC (it is not, of course, restricted to small devices). With an emulation layer, it should be possible to run programs written for this specification on top of RTSJ. The guidelines are:

- High-integrity profile.
- Easy to implement.
- Low runtime overhead.
- No syntactic extension of Java.
- Support for time measurement if WCET analysis tools are not available.
- Known overhead: Documentation of runtime behavior and memory requirements of every JVM operation and all methods has to be provided.
- Implementation possible on top of SEDC.

Application Structure: The following restrictions apply to the application:

- Initialization and mission phase.
- Fixed number of threads.
- Threads are created at initialization phase.
- All shared objects are allocated at initialization.

Threads: Three schedulable objects are defined. RtThread represents a periodic task. As usual, task work is coded in run() which gets called on missionStart(). A scoped memory object can be attached to an RtThread at creation. HwEvent represents an interrupt with a minimum interarrival time. If the hardware generates more interrupts, they get lost. SwEvent represents a software-generated event. It is triggered by fire() and needs to override handle(). Figure 2 shows the definition of the basic classes. An example of a real-time thread can be found in Figure 3.

Scheduling: Threads and events are scheduled with fixed priority. No real-time threads or events are scheduled during the initialization phase. Most real-time systems use a fixed-priority preemptive scheduler. Tasks with the same priority are usually scheduled in a FIFO order. Two common ways to assign priorities are rate monotonic or, 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 and we do not 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. With objects for which the priority is not set, a top priority is assumed. This avoids priority inversions on objects that are not accessible from the application (e.g. objects inside a library). In addition, the scheduler contains methods for worst-case time measurement for both the periodic work and handler methods. These measured execution times can be used during development when no WCET analysis tool is available.

```
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()
}
```

Figure 2: Schedulable objects

Memory: The profile does not support a garbage collector. All memory allocation should be done at initialization phase. For new objects during the mission phase, a scoped memory is provided. Every scoped memory area can be assigned to one RtThread. A scoped memory cannot be shared between threads. No references from the heap to scoped memory are allowed. Scoped memory is explicitly

entered and left with calls from the application logic. Memory areas are cleared on creation and when leaving the scope (call of exitMemory()) leading to a memory area with constant allocation time.

```
public class Worker extends RtThread {

    private SwEvent event;

    public Worker(int p, int t,
                  SwEvent ev) {
        super(p, t,
            // create a scoped memory area
            new Memory(10000)
        );
        event = ev;
    }

    public void run() {

        for (;;) {
            work();       // do some work
            event.fire(); // and fire an event
            // some work in scoped memory
            enterMemory();
            workWithMem();
            exitMemory();

            // wait for next period
            if (!waitForNextPeriod()) {
                missedDeadline();
            }
        }
        // should never reach this point
    }
}
```

Figure 3: A periodic real-time thread

Restriction of Java: Only WCET analyzable language constructs are allowed. Since the definition when to call the static class initializer is problematic, they are disallowed. This code has to be moved to a static method (e.g. init()) and called in the initialization phase. finalize() has a weak definition in Java. Because real-time systems run *forever*, objects in the heap, that is implicit *immortal* memory in this specification, will never be finalized. Objects in scoped memory are released on exitMemory(). However, finalizations on these objects complicate WCET analysis of exitMemory(). Due to the implementation and WCET analysis complexity dynamic class loading is avoided. A program analysis tool can greatly help in enforcing these restrictions.

6. Implementation Results

The proposed profile is implemented on JOP. In this section, the implementation of the simple real-time profile is compared with the RI (Reference Implementation) of RTSJ on top of Linux. The RI is an interpreting implementation of the JVM not optimized for performance. A commercial version of the RTSJ, JTime by TimeSys, should perform better. However, it was not possible to get a license of JTime for research purpose. JOP is implemented in a low cost FPGA (Cyclone EP1C6) from Altera clocked with 100 MHz. The test results for the RI were obtained on an Intel Pentium MMX 266 MHz, running Linux SuSE 8.2 with the TimeSys GPL Linux 3.1 [14] kernel as recommended by the RI. For each test 500 measurements were made.

Many activities in real-time systems must be performed periodically. Low release jitter is of major importance for tasks such as control loops. The test setting is similar to the periodic thread test in [15]. A single real-time thread only calls waitForNextPeriod() in a loop and records the time between calls that follow. Table 1 shows the average, standard deviation and extreme values for different period times on JOP. In Table 2 the same values are shown for the RI.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
T=80 us	80 us	28 us	52 us	115 us
T=100 us	100 us	0 us	100 us	100 us
T=500 us	500 us	0 us	500 us	500 us

Table 1: Jitter of Periodic Threads with JOP.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
T=5 ms	4.00 ms	7.92 ms	0.017 ms	19.90 ms
T=10 ms	6.64 ms	9.34 ms	0.019 ms	19.94 ms
T=20 ms	20.0 ms	0.015 ms	19.87 ms	20.14 ms
T=30 ms	30.0 ms	0.031 ms	29.69 ms	30.31 ms
T=35 ms	35.0 ms	5.001 ms	29.75 ms	40.25 ms

Table 2: Jitter of Periodic Threads with RI/RTSJ.

Using of a microsecond accurate timer interrupt, programmed by the scheduler, results in excellent performance of periodic threads in JOP. No jitter can be observed with a single thread at periods above 100 us. The RI performs inaccurately at periods below 20 ms. Larger periods that are multiples of 10 ms have very low jitter. However, using a period such as 35 ms shows a standard deviation of five ms. A detailed look at the collected samples shows only values of 30 and 40 ms. This implies a timer tick of 10 ms in the underlying operating system.

Table 3 gives the time for the context switch in processor clock cycles. The test setting consists of two threads. A low priority thread continuously stores the current time in a shared variable. A high priority periodic thread measures the time difference between this value and the time immediately after waitForNextPeriod().

	Avg.	*Std. Dev.*	*Min.*	*Max.*
JOP	4088	10.29	4083	4116
RI Linux	12923	1145	11529	21090

Table 3: Time for a Thread Switch in Clock Cycles.

In the next test setting, a high priority event handler is triggered by a low priority periodic thread. As

AsynchEventHandler performs not very well [15], a BoundAsynchEventHandler is used for the RI test program. The time elapsed between the invocation of fire() and the first statement of the event handler is measured. Table 4 shows the elapsed time in clock cycles for JOP and the RTSJ RI.

	Avg.	*Std. Dev.*	*Min.*	*Max.*
JOP	4283	3.0	4283	4350
RI Linux	69273	7832	63060	101292

Table 4: Dispatch Latency of Event Handlers.

The time to dispatch an asynchronous event is similar to the context switch time in JOP. The maximum value occurred only on the first event, all following events where dispatched with the minimum time. In the RI the dispatch time is about 6 times larger than a context switch. This indicates that the implementation of fire() and the communication of the event to the underlying operating system is not optimal.

To verify that the profile is expressive enough for high-integrity applications, Ravenscar Java was implemented on top of it. The only restriction observed is the absence of support for long in the JVM. This type is mandatory in the RTSJ for absolute and relative time definitions.

JOP is implemented on a low cost FPGA (EP1C6) consuming about 30% of the chip area. The unused area can be used for custom peripherals or a multi processor solution on a single chip. This resource usage is similar to Alteras NIOS, a 32-bit load/store RISC processor. The basic JVM with the proposed real-time extension consumes 9 KB of memory, leaving enough room for the application code and data even in very small devices. The real-time scheduler needs 0.5 KB per thread at runtime.

7. Conclusion

Some definitions of Java for embedded and real-time systems do exist. CLDC, as a restriction for embedded systems, suits small systems best, but is still large. The most common specification for real-time Java is the RTSJ. However, this specification is complex and large, making it not the primary choice for small embedded systems or high-integrity systems. Restrictions on the RTSJ can transform the definition into a high-integrity profile, but they inherit the complex API. A very small *configuration* for embedded systems and a high-integrity real-time *profile* that fits this configuration are proposed. The configuration and the profile are implemented on top of JOP and used in a number of real-world applications. Future work will explore additional hardware support for real-time systems in JOP. The implementation of JOP and the real-time profile are available at: http://www.jopdesign.com.

References

[1] E. Quinn and C. Christiansen. Java Technology Pays Positively. IDC Bulletin #W16212, May 1998.

[2] K. Nilsen. Issues in the Design and Implementation of Real-Time Java, July 1996. Published June 1996 in *Java Developers Journal*, republished in Q1 1998 *Real-Time Magazine*

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

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

[5] J. Gosling, B. Joy, G. Steele and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 1997.

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

[7] K. Nilsen, L. Carnahan and M. Ruark , editors. Requirements for Real-Time Extensions for the Java Platform. Published by *National Institute of Standards and Technology*. September 1999. Available at http://www.nist.gov/rt-java.

[8] B. Brosgol and B. Dobbing. Real-time convergence of Ada and Java. In *Proc. of the 2001 annual ACM SIGAda international conference on Ada*, pp.11-26, Bloomington, MN, 2001

[9] International J Consortium Specification. Real-Time Core Extensions, Draft 1.0.14, September 2nd 2000. Available at http://www.j-consortium.org/

[10] TimeSys. Real-Time Specification for Java Reference Implementation. http://www.timesys.com/

[11] 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

[12] A. Burns and B. Dobbing. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Proc. of the 1998 annual ACM SIGAda international conference on Ada*, pp. 1-6, Washington, D.C., United States, 1998

[13] 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

[14] TimeSys. Linux RTOS Standard Edition available at: http://www.timesys.com/

[15] A. Corsaro, D. Schmidt. Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems. Appeared at *The 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.