A Hardware Abstraction Layer in Java

MARTIN SCHOEBERL, Vienna University of Technology
STEPHAN KORSHOLM, Aalborg University
TOMAS KALIBERA, Purdue University
ANDERS P. RAVN, Aalborg University

Embedded systems use specialized hardware devices to interact with their environment, and since they have to be dependable, it is attractive to use a modern, type-safe programming language like Java to develop programs for them. Standard Java, as a platform-independent language, delegates access to devices, direct memory access, and interrupt handling to some underlying operating system or kernel, but in the embedded systems domain resources are scarce and a Java Virtual Machine (JVM) without an underlying middleware is an attractive architecture. The contribution of this article is a proposal for Java packages with hardware objects and interrupt handlers that interface to such a JVM. We provide implementations of the proposal directly in hardware, as extensions of standard interpreters, and finally with an operating system middleware. The latter solution is mainly seen as a migration path allowing Java programs to coexist with legacy system components. An important aspect of the proposal is that it is compatible with the Real-Time Specification for Java (RTSJ).

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Input/output*

General Terms: Languages, Design

Additional Key Words and Phrases: Device driver, embedded system, Java, Java virtual machine

ACM Reference Format:
Schoeberl, M., Korsholm, S., Kalibera, T., and Ravn, A. P. 2011. A hardware abstraction layer in Java. ACM Trans. Embed. Comput. Syst. 10, 4, Article 42 (November 2011), 40 pages.
DOI = 10.1145/2043662.2043666 http://doi.acm.org/10.1145/2043662.2043666

1. INTRODUCTION

When developing software for an embedded system, for instance an instrument, it is necessary to control specialized hardware devices, for instance a heating element or an interferometer mirror. These devices are typically interfaced to the processor through device registers and may use interrupts to synchronize with the processor. In order to make the programs easier to understand, it is convenient to introduce a *Hardware Abstraction Layer* (HAL), where access to device registers and synchronization through interrupts are hidden from conventional program components. A HAL defines an interface in terms of the constructs of the programming language used to develop the

The research leading to these results received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).
Authors' addresses: M. Schoeberl (corresponding author), Vienna University of Technology, Austria; email: mschoebe@mail.tuwien.ac.at; S. Korsholm, Aalborg University, Denmark; T. Kalibera, Purdue University, U.S.A.; A. P. Ravn, Aalborg University, Denmark.

DOI 10.1145/2043662.2043666 http://doi.acm.org/10.1145/2043662.2043666

application. Thus, the challenge is to develop an abstraction that gives efficient access to the hardware, while staying within the computational model provided by the programming language.

Our first ideas on a HAL for Java have been published in Schoeberl et al. [2008] and Korsholm et al. [2008]. This article combines the two papers, provides a much wider background of related work, gives two additional experimental implementations, and gives performance measurements that allow an assessment of the efficiency of the implementations. The remainder of this section introduces the concepts of the Java-based HAL.

1.1 Java for Embedded Systems

Over the nearly 15 years of its existence Java has become a popular programming language for desktop and server applications. The concept of the Java Virtual Machine (JVM) as the execution platform enables portability of Java applications. The language, its API specification, as well as JVM implementations have matured; Java is today employed in large-scale industrial applications. The automatic memory management takes away a burden from the application programmers and together with type safety helps to isolate problems and, to some extent, even run untrusted code. It also enhances security; attacks like stack overflow are not possible. Java integrates threading support and dynamic loading into the language, making these features easily accessible on different platforms. The Java language and JVM specifications are proven by different implementations on different platforms, making it relatively easy to write platform-independent Java programs that run on different JVM implementations and underlying OS/hardware. Java has a standard API for a wide range of libraries, the use of which is thus again platform-independent. With the ubiquity of Java, it is easy to find qualified programmers who know the language, and there is strong tool support for the whole development process. According to an experimental study [Phipps 1999], Java has lower bug rates and higher productivity rates than C++. Indeed, some of these features come at a price of larger footprint (the virtual machine is a nontrivial piece of code), typically higher memory requirements, and sometimes degraded performance, but this cost is accepted in industry.

Recent real-time Java virtual machines based on the Real-Time Specification for Java (RTSJ) provide controlled and safe memory allocation. Also there are platforms for less critical systems with real-time garbage collectors. Thus, Java is ready to make its way into the embedded systems domain. Mobile phones, PDAs, or set-top boxes run Java Micro Edition, a Java platform with a restricted set of standard Java libraries. Real-time Java has been and is being evaluated as a potential future platform for space avionics both by NASA and ESA space agencies. Some Java features are even more important for embedded than for desktop systems because of missing features of the underlying platform. For instance, the RTEMS operating system used by ESA for space missions does not support hardware memory protection even for CPUs that do support it (like LEON3, a CPU for ESA space missions). With Java's type safety hardware protection is not needed to spatially isolate applications. Moreover, RTEMS does not support dynamic libraries, but Java can load classes dynamically.

Many embedded applications require very small platforms, therefore it is interesting to remove as much as possible of an underlying operating system or kernel, where a major part of code is dedicated to handling devices. Furthermore, Java is considered as the future language for safety-critical systems [Henties et al. 2009]. As certification of safety-critical systems is very expensive, the usual approach is to minimize the code base and supporting tools. Using two languages (e.g., C for programming device handling in the kernel and Java for implementing the processing of data) increases

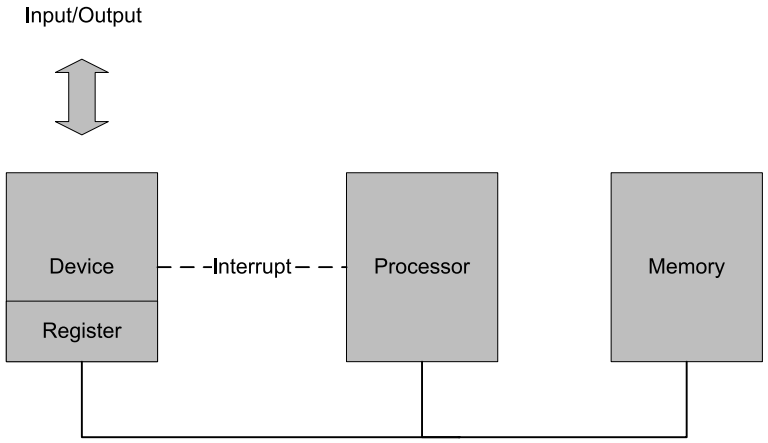

Fig. 1. The hardware: a bus connects a processor to device registers and memory, and an interrupt bus connects devices to a processor.

the complexity of generating a safety case. A Java-only system reduces the complexity of the tool support and therefore the certification effort. Even in less critical systems the same issues will show up as decreased productivity and dependability of the software. Thus it makes sense to investigate a general solution that interfaces Java to the hardware platform; that is the objective of the work presented here.

1.2 Hardware Assumptions

The hardware platform is built up along one or more buses (in small systems typically only one) that connect the processor with memory and device controllers. Device controllers have reserved some part of the address space of a bus for its device registers. They are accessible for the processor as well, either through special I/O instructions or by ordinary instructions when the address space is the same as the one for addressing memory, a so called memory mapped I/O solution. In some cases the device controller will have Direct Memory Access (DMA) as well, for instance for high-speed transfer of blocks of data. Thus the basic communication paradigm between a controller and the processor is shared memory through the device registers and/or through DMA. With these facilities only, synchronization has to be done by testing and setting flags, which means that the processor has to engage in some form of busy waiting. This is eliminated by extending the system with an interrupt bus, where device controllers can generate a signal that interrupts the normal flow of execution in the processor and direct it to an interrupt handling program. Since communication is through shared data structures, the processor and the controllers need a locking mechanism; therefore interrupts can be enabled or disabled by the processor through an interrupt control unit. The typical hardware organization is summarized in Figure 1.

1.3 A Computational Model

In order to develop a HAL, the device registers and interrupt facilities must be mapped to programming language constructs, such that their use corresponds to the computational model underlying the language. In the following we give simple device examples which illustrate the solution we propose for doing it for Java.

1.3.1 Hardware Objects. Consider a simple Parallel Input/Output (PIO) device controlling a set of input and output pins. The PIO uses two registers: the *data register* and

```
public final class ParallelPort {
    public volatile int data;
    public volatile int control;
}

int inval, outval;
myport = JVMMechanism.getParallelPort();
...
inval = myport.data;
myport.data = outval;
```

Fig. 2. The parallel port device as a simple Java class.

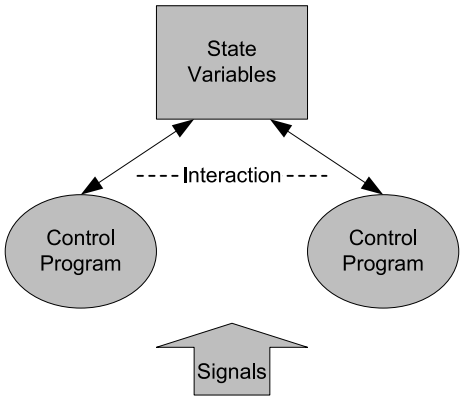

Fig. 3. Computational model: several threads of execution communicate via shared state variables and receive signals.

the *control register*. Writing to the data register stores the value into an internal latch that drives the output pins. Reading from the data register returns the value that is present on the input pins. The control register configures the direction for each PIO pin. When bit n in the control register is set to 1, pin n drives out the value of bit n of the data register. A 0 at bit n in the control register configures pin n as input pin. At reset the port is usually configured as input port, a safe default configuration.

In an object-oriented language the most natural way to represent a device is as an object: the *hardware object*. Figure 2 shows a class definition, object instantiation, and use of the hardware object for the simple parallel port. An instance of the class ParallelPort is the hardware object that represents the PIO. The reference myport points to the hardware object. To provide this convenient representation of devices as objects, a JVM internal mechanism is needed to access the device registers via object fields and to *create* the device object and receive a reference to it. We elaborate on the idea of hardware objects in Section 3.1 and present implementations in Section 4.

1.3.2 Interrupts. When we consider an interrupt, it must invoke some program code in a method that handles it. We need to map the interruption of normal execution to some language concept, and here the concept of an asynchronous event is useful. The resulting computational model for the programmer is shown in Figure 3. The signals are external, asynchronous events that map to interrupts.

A layered implementation of this model with a kernel close to the hardware and applications on top has been very useful in general-purpose programming. Here one may even extend the kernel to manage resources and provide protection mechanisms such that applications are safe from one another, as for instance when implementing

```
public class RS232ReceiveInterruptHandler extends InterruptHandler {
    private RS232 rs232;
    private InterruptControl interruptControl;

    private byte UartRxBuffer[];
    private short UartRxWrPtr;

    ...

    protected void handle() {

        synchronized(this) {
            UartRxBuffer[UartRxWrPtr++] = rs232.P0_UART_RX_TX_REG;
            if (UartRxWrPtr >= UartRxBuffer.length) UartRxWrPtr = 0;
        }
        rs232.P0_CLEAR_RX_INT_REG = 0;
        interruptControl.RESET_INT_PENDING_REG = RS232.CLR_UART_RX_INT_PENDING;
    }
}
```

Fig. 4. An example interrupt handler for an RS232 interface. On an interrupt the method handle() is invoked. The private objects rs232 and interruptControl are hardware objects that represent the device registers and the interrupt control unit.

trusted interoperable computing platforms [Group 2008]. Yet there is a price to pay which may make the solution less suitable for embedded systems: adding new device drivers is an error-prone activity [Chou et al. 2001], and protection mechanisms impose a heavy overhead on context switching when accessing devices.

The alternative we propose is to use Java directly since it already supports multithreading and use methods in the special InterruptHandler objects to handle interrupts. The idea is illustrated in Figure 4, and the details, including synchronization and interaction with the interrupt control, are elaborated in Section 3.2. Implementations are found in Section 4.

1.4 Mapping Between Java and the Hardware

The proposed interfacing from hardware to Java does not require language extensions. The Java concepts of packages, classes, and synchronized objects turn out to be powerful enough to formulate the desired abstractions. The mapping is done at the level of the JVM. The JVM already provides typical OS functions handling:

— address space and memory management;
— thread management;
— interprocess communication.

These parts need to be modified so they cater for interfaces to the hardware.

Yet, the architectures of JVMs for embedded systems are more diverse than on desktop or server systems. Figure 5 shows variations of Java implementations in embedded systems and an example of the control flow for a Web server application. The standard approach with a JVM running on top of an Operating System (OS) is shown in Figure 5(a).

A JVM without an OS is shown in Figure 5(b). This solution is often called *running on the bare metal*. The JVM acts as the OS and provides thread scheduling and low-level access to the hardware. In this case the network stack can be written entirely in

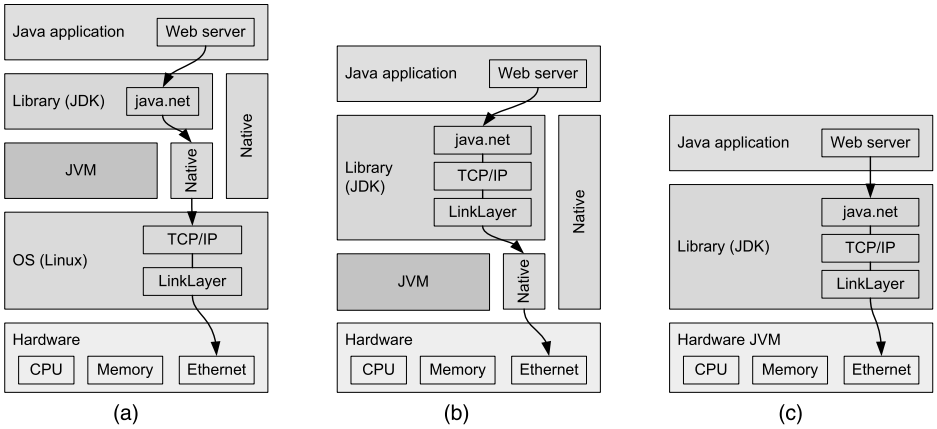

Fig. 5. Configurations for an embedded JVM: (a) standard layers for Java with an operating system, equivalent to desktop configurations; (b) a JVM on the bare metal; and (c) a JVM as a Java processor.

Java. JNode[1] is an approach to implement the OS entirely in Java. This solution has become popular even in server applications.[2]

Figure 5(c) shows an embedded solution where the JVM is part of the hardware layer: it is implemented in a Java processor. With this solution the native layer can be completely avoided and all code (application and system code) is written entirely in Java.

Figure 5 shows also the data and control flow from the application down to the hardware. The example consists of a Web server and an Internet connection via Ethernet. In case (a) the application Web server talks with java.net in the Java library. The flow goes down via a native interface to the TCP/IP implementation and the link-layer device driver within the OS (usually written in C). The device driver talks with the Ethernet chip. In (b) the OS layer is omitted: the TCP/IP layer and the link-layer device driver are now part of the Java library. In (c) the JVM is part of the hardware layer, and direct access from the link-layer driver to the Ethernet hardware is mandatory.

With our proposed HAL, as shown in Figure 6, the native interface within the JVM in (a) and (b) disappears. Note how the network stack moves up from the OS layer to the Java library in example (a). All three versions show a pure Java implementation of the whole network stack. The Java code is the same for all three solutions. Version (b) and (c) benefit from hardware objects and interrupt handlers in Java as access to the Ethernet device is required from Java source-code. In Section 5 we show a simple Web server application implemented completely in Java as evaluation of our approach.

1.5 Contributions

The key contribution of this article is a proposal for a Java HAL that can run on the bare metal while still being *safe*. This idea is investigated in quite a number of places which are discussed in the related work section where we comment on our initial ideas as well. In summary, the proposal gives an interface to hardware that has the following benefits.

[1]`http://www.jnode.org/`
[2]BEA System offers the JVM LiquidVM that includes basic OS functions and does not need a guest OS.

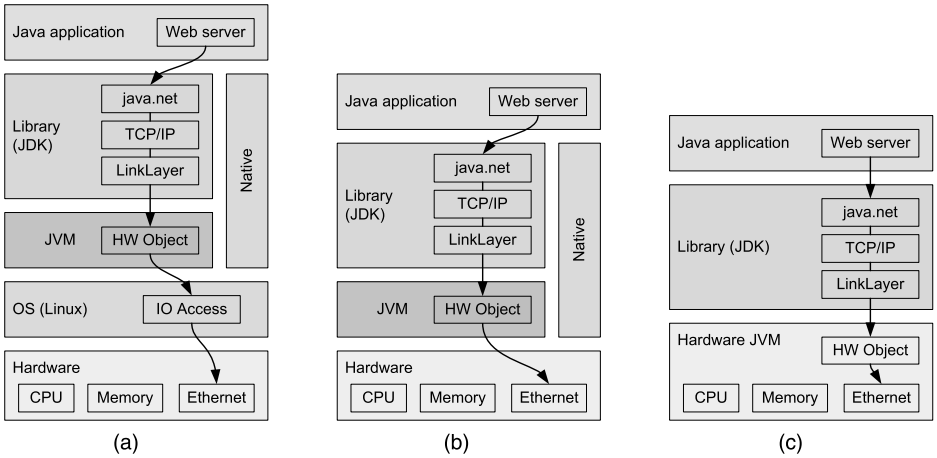

Fig. 6. Configurations for an embedded JVM with hardware objects and interrupt handlers: (a) standard layers for Java with an operating system, equivalent to desktop configurations; (b) a JVM on the bare metal; and (c) a JVM as a Java processor.

Object-oriented. An object representing a device is the most natural integration into an object-oriented language, and a method invocation to a synchronized object is a direct representation of an interrupt.

Safe. The safety of Java is not compromised. Hardware objects map object fields to device registers. With a correct class that represents the device, access to it is safe. Hardware objects can be created only by a factory residing in a special package.

Generic. The definition of a hardware object and an interrupt handler is independent of the JVM. Therefore, a common standard for different platforms can be defined.

Efficient. Device register access is performed by single bytecodes getfield and putfield. We avoid expensive native calls. The handlers are first-level handlers; there is no delay through event queues.

The proposed Java HAL would not be useful if it had to be modified for each particular kind of JVM; thus a second contribution of this article is a number of prototype implementations illustrating the architectures presented in Figure 6: implementations in Kaffe [Wilkinson 1996] and OVM [Armbruster et al. 2007] represent the architecture with an OS (Figure 6(a)), the implementation in SimpleRTJ [RTJ Computing 2000] represents the bare metal solution (Figure 6(b)), and the implementation in JOP [Schoeberl 2008] represents the Java processor solution (Figure 6(c)).

Finally, we must not forget the claim for efficiency, and therefore the article ends with some performance measurements that indicate that the HAL layer is generally as efficient as native calls to C code external to the JVM.

2. RELATED WORK

Already in the 1970's it was recognized that an operating system might not be the optimal solution for special-purpose applications. Device access was integrated into high-level programming languages like Concurrent Pascal [Hansen 1977; Ravn 1980] and Modula (Modula-2) [Wirth 1977, 1982] along with a number of similar languages, for example, UCSD Pascal. They were meant to eliminate the need for operating systems and were successfully used in a variety of applications. The programming language Ada, which has been dominant in defence and space applications to this day,

may be seen as a continuation of these developments. The advent of inexpensive microprocessors, from the mid 1980's and on, lead to a regression to assembly and C programming. The hardware platforms were small with limited resources and the developers were mostly electronic engineers, who viewed them as electronic controllers. Program structure was not considered a major issue in development. Nevertheless, the microcomputer has grown, and is now far more powerful than the minicomputer that it replaced. With powerful processors and an abundance of memory, the ambitions for the functionality of embedded systems grow, and programming becomes a major issue because it may turn out to be the bottleneck in development. Consequently, there is a renewed interest in this line of research.

An excellent overview of historical solutions to access hardware devices from and implement interrupt handlers in high-level languages, including C, is presented in Chapter 15 of Burns and Wellings [2001]. The solution to device register access in Modula-1 (Chapter 15.3) is very much like C; however, the constructs are safer because they are encapsulated in modules. Interrupt handlers are represented by threads that block to wait for the interrupt. In Ada (Chapter 15.4) the representation of individual fields in registers can be described precisely by *representation* classes, while the corresponding structure is bound to a location using the Address attribute. An interrupt is represented in the current version of Ada by a protected procedure, although initially represented (Ada 83) by task entry calls.

The main ideas in having device objects are thus found in the earlier safe languages, and our contribution is to align them with a Java model, and in particular, as discussed in Section 4, implementation in a JVM. From the Ada experience we learn that direct handling of interrupts is a desired feature.

2.1 The Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) [Bollella et al. 2000] defines a JVM extension which allows better timeliness control compared to a standard JVM. The core features are: fixed priority scheduling, monitors which prevent priority inversion, scoped memory for objects with limited lifetime, immortal memory for objects that are never finalized, and asynchronous events with CPU time consumption control.

The RTSJ also defines an API for direct access to physical memory, including hardware registers. Essentially one uses RawMemoryAccess at the level of primitive data types. Although the solution is efficient, this representation of physical memory is not object-oriented, and there are some safety issues: When one raw memory area represents an address range where several devices are mapped to, there is no protection between them. Yet, a type-safe layer with support for representing individual registers can be implemented on top of the RTSJ API.

The RTSJ specification suggests that asynchronous events are used for interrupt handling. Yet, it neither specifies an API for interrupt control nor semantics of the handlers. Any interrupt handling application thus relies on some proprietary API and proprietary event handler semantics. Second-level interrupt handling can be implemented within the RTSJ with an AsyncEvent that is bound to a *happening*. The happening is a string constant that represents an interrupt, but the meaning is implementation-dependent. An AsyncEventHandler or BoundAsyncEventHandler can be added as handler for the event. Also an AsyncEventHandler can be added via a POSIXSignalHandler to handle POSIX signals. An interrupt handler, written in C, can then use one of the two available POSIX user signals.

RTSJ offers facilities very much in line with Modula or Ada for encapsulating memory-mapped device registers. However, we are not aware of any RTSJ implementation that implements RawMemoryAccess and AsyncEvent with support for low-level

device access and interrupt handling. Our solution could be used as specification of such an extension. It would still leave the first-level interrupt handling hidden in an implementation; therefore an interesting idea is to define and implement a two-level scheduler for the RTSJ. It should provide the first-level interrupt handling for asynchronous events bound to interrupts and delegate other asynchronous events to an underlying second-level scheduler, which could be the standard fixed priority preemptive scheduler. This would be a fully RTSJ-compliant implementation of our proposal.

2.2 Hardware Interface in JVMs

The aJile Java processor [aJile 2000] uses native functions to access devices. Interrupts are handled by registering a handler for an interrupt source (e.g., a GPIO pin). Systronix suggests[3] to keep the handler short, as it runs with interrupts disabled, and delegate the real handling to a thread. The thread waits on an object with ceiling priority set to the interrupt priority. The handler just notifies the waiting thread through this monitor. When the thread is unblocked and holds the monitor, effectively all interrupts are disabled.

Komodo [Kreuzinger et al. 2003] is a multithreaded Java processor targeting real-time systems. On top of the multiprocessing pipeline the concept of interrupt service threads is implemented. For each interrupt one thread slot is reserved for the interrupt service thread. It is unblocked by the signaling unit when an interrupt occurs. A dedicated thread slot on a fine-grain multithreading processor results in a very short latency for the interrupt service routine. No thread state needs to be saved. However, this comes at the cost to store the complete state for the interrupt service thread in the hardware. In the case of Komodo, the state consists of an instruction window and the on-chip stack memory. Devices are represented by Komodo-specific I/O classes.

Muvium [Caska 2009] is an ahead-of-time compiling JVM solution for very resource-constrained microcontrollers (Microchip PIC). Muvium uses an Abstract Peripheral Toolkit (APT) to represent devices. APT is based on an event-driven model for interaction with the external world. Device interrupts and periodic activations are represented by events. Internally, events are mapped to threads with priority dispatched by a preemptive scheduler. APT contains a large collection of classes to represent devices common in embedded systems.

In summary, access to device registers is handled in both aJile, Komodo, and Muvium by abstracting them into library classes with access methods. This leaves the implementation to the particular JVM and does not give the option of programming them at the Java level. It means that extension with new devices involves programming at different levels, which we aim to avoid. Interrupt handling in aJile is essentially first level, but with the twist that it may be interpreted as RTSJ event handling, although the firing mechanism is atypical. Our mechanism would free this binding and allow other forms of programmed notification, or even leaving out notification altogether. Muvium follows the line of RTSJ and has a hidden first-level interrupt handling. Komodo has a solution with first-level handling through a full context switch; this is very close to the solution advocated in Modula 1, but it has in general a larger overhead than we would want to incur.

2.3 Java Operating Systems

The JX Operating System [Felser et al. 2002] is a microkernel system written mostly in Java. The system consists of components which run in *domains*, each domain

[3]A template can be found at `http://practicalembeddedjava.com/tutorials/aJileISR.html`.

having its own garbage collector, threads, and a scheduler. There is one global preemptive scheduler that schedules the domain schedulers which can be both preemptive and nonpreemptive. Inter-domain communication is only possible through communication channels exported by services. Low-level access to the physical memory, memory-mapped device registers, and I/O ports are provided by the core ("zero") domain services, implemented in C. At the Java level ports and memory areas are represented by objects, and registers are methods of these objects. Memory is read and written by access methods of Memory objects. Higher layers of Java interfaces provide type-safe access to the registers; the low-level access is not type safe.

Interrupt handlers in JX are written in Java and are run through portals; they can reside in any domain. Interrupt handlers cannot interrupt the garbage collector (the GC disables interrupts), run with CPU interrupts disabled, must not block, and can only allocate a restricted amount of memory from a reserved per-domain heap. Execution time of interrupt handlers can be monitored: on a deadline violation the handler is aborted and the interrupt source disabled. The first-level handlers can unblock a waiting second-level thread either directly or via setting a state of a AtomicVariable synchronization primitive.

The Java New Operating System Design Effort (JNode[4]) [Lohmeier 2005] is an OS written in Java where the JVM serves as the OS. Drivers are written entirely in Java. Device access is performed via native function calls. A first-level interrupt handler, written in assembler, unblocks a Java interrupt thread. From this thread the device driver-level interrupt handler is invoked with interrupts disabled. Some device drivers implement a synchronized handleInterrupt(int irq) and use the driver object to signal the upper layer with notifyAll(). During garbage collection all threads are stopped including the interrupt threads.

The Squawk VM [Simon et al. 2006], now available open-source,[5] is a platform for wireless sensors. Squawk is mostly written in Java and runs without an OS. Device drivers are written in Java and use a form of peek and poke interface to access the device registers. Interrupt handling is supported by a device driver thread that waits for an event from the JVM. The first-level handler, written in assembler, disables the interrupt and notifies the JVM. On a rescheduling point the JVM resumes the device driver thread. It has to reenable the interrupt. The interrupt latency depends on the rescheduling point and on the activity of the garbage collector. For a single device driver thread an average case latency of 0.1 ms is reported. For a realistic workload with an active garbage collector a worst-case latency of 13 ms has been observed.

Our proposed constructs should be able to support the Java operating systems. For JX we observe that the concepts are very similar for interrupt handling, and actually for device registers as well. A difference is that we make device objects distinct from memory objects which should give better possibilities for porting to architectures with separate I/O buses. JNode is more traditional and hides first-level interrupt handling and device accesses in the JVM, which may - be less portable than our implementation. The Squawk solution has to have a very small footprint, but on the other hand it can probably rely on having few devices. Device objects would be at least as efficient as the peeks and pokes, and interrupt routines may eliminate the need for multithreading for simple systems, for example, with cyclic executives. Overall, we conclude that our proposed constructs will make implementation of a Java OS more efficient and perhaps more portable.

[4]http://jnode.org/
[5]https://squawk.dev.java.net/

2.4 TinyOS and Singularity

TinyOS [Hill et al. 2000] is an operating system designed for low-power, wireless sensor networks. TinyOS is not a a traditional OS, but provides a framework of components that are linked with the application code. The component-based programming model is supported by nesC [Gay et al. 2003], a dialect of C. TinyOS components provide following abstractions: *commands* represent requests for a service of a component; *events* signal the completion of a service; and *tasks* are functions executed nonpreemptive by the TinyOS scheduler. Events also represent interrupts and preempt tasks. An event handler may post a task for further processing, which is similar to a second-level interrupt handler.

I/O devices are encapsulated in components and the standard distribution of TinyOS includes a rich set of standard I/O devices. A Hardware Presentation Layer (HPL) abstracts the platform-specific access to the hardware (either memory or port mapped). Our proposed HAL is similar to the HPL, but represents the I/O devices as Java objects. A further abstraction into I/O components can be built above our presented Java HAL.

Singularity [Hunt et al. 2005] is a research OS based on a runtime managed language (an extension of C#) to build a software platform with the main goal to be dependable. A small HAL (IoPorts, IoDma, IoIrq, and IoMemory) provides access to PC hardware. C# style attributes (similar to Java annotations) on fields are used to define the mapping of class fields to I/O ports and memory addresses. The Singularity OS clearly uses device objects and interrupt handlers, thus demonstrating that the ideas presented here transfer to a language like C#.

2.5 Summary

In our analysis of related work we see that our contribution is a selection, adaptation, refinement, and implementation of ideas from earlier languages and platforms for Java. A crucial point, where we have spent much time, is to have a clear interface between the Java layer and the JVM. Here we have used the lessons from the Java OS and the JVM interfaces. Finally, it has been a concern to be consistent with the RTSJ because this standard and adaptations of it are the instruments for developing embedded real-time software in Java.

3. THE HARDWARE ABSTRACTION LAYER

In the following section the hardware abstraction layer for Java is defined. Low-level access to devices is performed via hardware objects. Synchronization with a device can be performed with interrupt handlers implemented in Java. Finally, portability of hardware objects, interrupt handlers, and device drivers is supported by a generic configuration mechanism.

3.1 Device Access

Hardware objects map object fields to device registers. Therefore, field access with bytecodes putfield and getfield accesses device registers. With a correct class that represents a device, access to it is safe; it is not possible to read or write to an arbitrary memory address. A memory area (e.g., a video frame buffer) represented by an array is protected by Java's array bounds check.

In a C-based system the access to I/O devices can either be represented by a C struct (similar to the class shown in Figure 2) for memory-mapped I/O devices or needs to be accessed by function calls on systems with a separate I/O address space. With the hardware object abstraction in Java the JVM can represent an I/O device as a class independent of the underlying low-level I/O mechanism. Furthermore, the strong

```
public abstract class HardwareObject {
    HardwareObject() {};
}
```

Fig. 7. The marker class for hardware objects.

```
public final class SerialPort extends HardwareObject {

    public static final int MASK_TDRE = 1;
    public static final int MASK_RDRF = 2;

    public volatile int status;
    public volatile int data;

    public void init(int baudRate) {...}
    public boolean rxFull() {...}
    public boolean txEmpty() {...}
}
```

Fig. 8. A serial port class with device methods.

typing of Java avoids hard to find programming errors due to wrong pointer casts or wrong pointer arithmetic.

All hardware classes have to extend the abstract class HardwareObject (see Figure 7). This empty class serves as type marker. Some implementations use it to distinguish between plain objects and hardware objects for the field access. The package-visible-only constructor disallows creation of hardware objects by the application code that resides in a different package. Figure 8 shows a class representing a serial port with a status register and a data register. The status register contains flags for receive register full and transmit register empty; the data register is the receive and transmit buffer. Additionally, we define device-specific constants (bit masks for the status register) in the class for the serial port. All fields represent device registers that can change due to activity of the hardware device. Therefore, they must be declared volatile.

In this example we have included some convenience methods to access the hardware object. However, for a clear separation of concerns, the hardware object represents only the device state (the registers). We do not add instance fields to represent additional state, that is, mixing device registers with heap elements. We cannot implement a complete device driver within a hardware object; instead a complete device driver owns a number of private hardware objects along with data structures for buffering, and it defines interrupt handlers and methods for accessing its state from application processes. For device-specific operations, such as initialization of the device, methods in hardware objects are useful.

Usually each device is represented by exactly one hardware object (see Section 3.3.1). However, there might be use cases where this restriction should be relaxed. Consider a device where some registers should be accessed by system code only and some other by application code. In JOP there is such a device: a system device that contains a 1 MHz counter, a corresponding timer interrupt, and a watchdog port. The timer interrupt is programmed relative to the counter and used by the real-time scheduler, a JVM internal service. However, access to the counter can be useful for the application code. Access to the watchdog register is required from the application level. The watchdog is used for a sign-of-life from the application. If not triggered every second the complete system is restarted. For this example it is useful to represent one

```
public final class SysCounter extends HardwareObject {

    public volatile int counter;
    public volatile int timer;
    public volatile int wd;
}

public final class AppCounter extends HardwareObject {

    public volatile int counter;
    private volatile int timer;
    public volatile int wd;
}

public final class AppGetterSetter extends HardwareObject {

    private volatile int counter;
    private volatile int timer;
    private volatile int wd;

    public int getCounter() {
        return counter;
    }

    public void setWd(boolean val) {
        wd = val ? 1 : 0;
    }
}
```

Fig. 9. System and application classes, one with visibility protection and one with setter and getter methods, for a single hardware device

hardware device by two different classes: one for system code and one for application code. We can protect system registers by private fields in the hardware object for the application. Figure 9 shows the two class definitions that represent the same hardware device for system and application code respectively. Note how we changed the access to the timer interrupt register to private for the application hardware object.

Another option, shown in class AppGetterSetter, is to declare all fields private for the application object and use setter and getter methods. They add an abstraction on top of hardware objects and use the hardware object to implement their functionality. Thus we still do not need to invoke native functions.

Use of hardware objects is straightforward. After obtaining a reference to the object all that has to be done (or can be done) is to read from and write to the object fields. Figure 10 shows an example of client code. The example is a *Hello World* program using low-level access to a serial port via a hardware object. Creation of hardware objects is more complex and described in Section 3.3. Furthermore, it is JVM-specific and Section 4 describes implementations in four different JVMs.

For devices that use DMA (e.g., video frame buffer, disk, and network I/O buffers) we map that memory area to Java arrays. Arrays in Java provide access to raw memory in an elegant way: the access is simple and safe due to the array bounds checking done by the JVM. Hardware arrays can be used by the JVM to implement higher-level abstractions from the RTSJ such as RawMemory or scoped memory.

Interaction between the Garbage Collector (GC) and hardware objects needs to be crafted into the JVM. We do not want to collect hardware objects. The hardware object

```
import com.jopdesign.io.*;

public class Example {

    public static void main(String[] args) {

        BaseBoard fact = BaseBoard.getBaseFactory();
        SerialPort sp = fact.getSerialPort();

        String hello = "Hello World!";

        for (int i=0; i<hello.length(); ++i) {
            // busy wait on transmit buffer empty
            while ((sp.status & SerialPort.MASK_TDRE) == 0)
                ;
            // write a character
            sp.data = hello.charAt(i);
        }
    }
}
```

Fig. 10. A "Hello World" example with low-level device access via a hardware object.

Table I. Dispatching Properties of Different ISR Strategies

ISR	Context switches	Priorities
Handler	2	Hardware
Event	3–4	Software

should not be scanned for references.[6] This is permissible when only primitive types are used in the class definition for hardware objects; the GC scans only reference fields. To avoid collecting hardware objects, we mark the object to be skipped by the GC. The type inheritance from HardwareObject can be used as the marker.

3.2 Interrupt Handling

An Interrupt Service Routine (ISR) can be integrated with Java in two different ways: as a first-level *handler* or a second-level *event* handler.

ISR handler. The interrupt is a method call initiated by the device.

Usually this abstraction is supported in hardware by the processor and called a first-level handler.

ISR event. The interrupt is represented by an asynchronous notification directed to a thread that is unblocked from a wait-state. This is also called deferred interrupt handling.

An overview of the dispatching properties of the two approaches is given in Table I. The ISR handler approach needs only two context switches and the priority is set by the hardware. With the ISR event approach, handlers are scheduled at software priorities. The initial first-level handler, running at hardware priority, fires the event

[6]If a hardware coprocessor, represented by a hardware object, itself manipulates an object on the heap and holds the only reference to that object it has to be scanned by the GC.

for the event handler. Also the first-level handler will notify the scheduler. In the best case three context switches are necessary: one to the first-level handler, one to the ISR event handler, and one back to the interrupted thread. If the ISR handler has a lower priority than the interrupted thread, an additional context switch from the first-level handler back to the interrupted thread is necessary.

Another possibility is to represent an interrupt as a thread that is released by the interrupt. Direct support by the hardware (e.g., the interrupt service thread in Komodo [Kreuzinger et al. 2003]) gives fast interrupt response times. However, standard processors support only the handler model directly.

Direct handling of interrupts in Java requires the JVM to be prepared to be interrupted. In an interpreting JVM an initial handler will reenter the JVM to execute the Java handler. A compiling JVM or a Java processor can directly invoke a Java method as response to the interrupt. A compiled Java method can be registered directly in the ISR dispatch table.

If an internal scheduler is used (also called *green threads*) the JVM will need some refactoring in order to support asynchronous method invocation. Usually JVMs control the rescheduling at the JVM level to provide a lightweight protection of JVM internal data structures. These preemption points are called pollchecks or yield points; also some or all can be GC preemption points. In fact the preemption points resemble cooperative scheduling at the JVM level and use priority for synchronization. This approach works only for uniprocessor systems; for multiprocessors explicit synchronization has to be introduced.

In both cases there might be critical sections in the JVM where reentry cannot be allowed. To solve this problem the JVM must disable interrupts around critical nonreenterable sections. The more fine-grained this disabling of interrupts can be done, the more responsive to interrupts the system will be.

One could opt for second-level handlers only. An interrupt fires and releases an associated schedulable object (handler). Once released, the handler will be scheduled by the JVM scheduler according to the release parameters. This is the RTSJ approach. The advantage is that interrupt handling is done in the context of a normal Java thread and scheduled as any other thread running on the system. The drawback is that there will be a delay from the occurrence of the interrupt until the thread gets scheduled. Additionally, the meaning of interrupt priorities, levels, and masks used by the hardware may not map directly to scheduling parameters supported by the JVM scheduler.

In the following we focus on the ISR handler approach, because it allows programming the other paradigms within Java.

3.2.1 Hardware Properties. We assume interrupt hardware as it is found in most computer architectures: interrupts have a fixed priority associated with them; they are set with a solder iron. Furthermore, interrupts can be globally disabled. In most systems the first-level handler is called with interrupts globally disabled. To allow nested interrupts (being able to interrupt the handler by a higher-priority interrupt as in preemptive scheduling) the handler has to enable interrupts again. However, to avoid priority inversion between handlers only interrupts with a higher priority will be enabled, either by setting the interrupt level or setting the interrupt mask. Software threads are scheduled by a timer interrupt and usually have a lower priority than interrupt handlers (the timer interrupt has the lowest priority of all interrupts). Therefore, an interrupt handler is never preempted by a software thread.

Mutual exclusion between an interrupt handler and a software thread is ensured by disabling interrupts: either all interrupts or selectively. Again, to avoid priority inversion, only interrupts of a higher priority than the interrupt that shares the data with

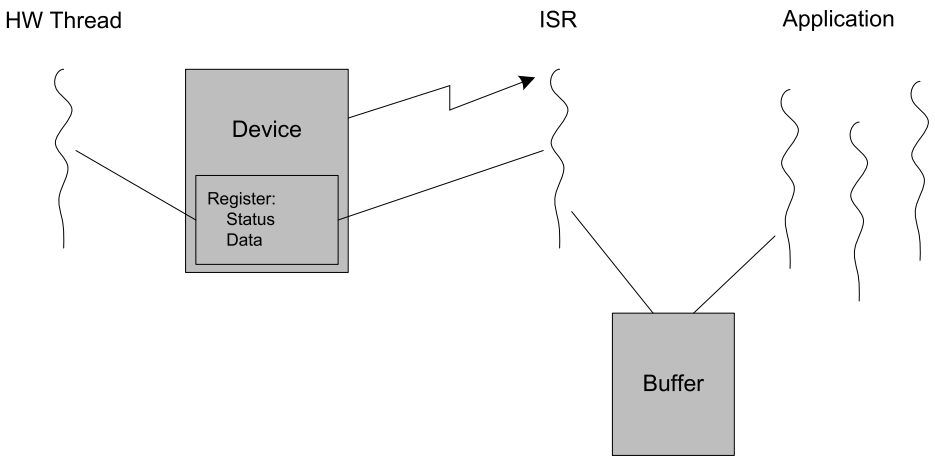

Fig. 11. Threads and shared data.

the software thread can be enabled. This mechanism is in effect the priority ceiling emulation protocol [Sha et al. 1990], sometimes called immediate ceiling protocol. It has the virtue that it eliminates the need for explicit locks (or Java monitors) on shared objects. Note that mutual exclusion with interrupt disabling works only in a uniprocessor setting. A simple solution for multiprocessors is to run the interrupt handler and associated software threads on the same processor core. A more involved scheme would be to use spin-locks between the processors.

When a device asserts an interrupt request line, the interrupt controller notifies the processor. The processor stops execution of the current thread. A partial thread context (program counter and processor status register) is saved. Then the ISR is looked up in the interrupt vector table and a jump is performed to the first instruction of the ISR. The handler usually saves additional thread context (e.g., the register file). It is also possible to switch to a new stack area. This is important for embedded systems where the stack sizes for all threads need to be determined at link time.

3.2.2 Synchronization. Java supports synchronization between Java threads with the synchronized keyword, either as a means of synchronizing access to a block of statements or to an entire method. In the general case this existing synchronization support is not sufficient to synchronize between interrupt handlers and threads.

Figure 11 shows the interacting active processes and the shared data in a scenario involving the handling of an interrupt. Conceptually three threads interact: (1) a hardware device thread representing the device activity, (2) the ISR, and (3) the application or device driver thread. These three share two types of data.

Device data. The hardware thread and ISR share access to the device registers of the device signaling the interrupt.

Application data. The ISR and application or device driver share access to, for example, a buffer conveying information about the interrupt to the application.

Regardless of which interrupt handling approach is used in Java, synchronization between the ISR and the device registers must be handled in an ad hoc way. In general there is no guarantee that the device has not changed the data in its registers; but if the ISR can be run to completion within the minimum inter-arrival time of the interrupt the content of the device registers can be trusted.

For synchronization between the ISR and the application (or device driver) the following mechanisms are available. When the ISR handler runs as a software thread, standard synchronization with object monitors can be used. When using the ISR handler approach, the handler is no longer scheduled by the normal Java scheduler, but by the hardware. While the handler is running, all other executable elements are suspended, including the scheduler. This means that the ISR cannot be suspended, must not block, or must not block via a language-level synchronization mechanism; the ISR must run to completion in order not to freeze the system. This means that when the handler runs, it is guaranteed that the application will not get scheduled. It follows that the handler can access data shared with the application without synchronizing with the application. As the access to the shared data by the interrupt handler is not explicitly protected by a synchronized method or block, the shared data needs to be declared volatile.

On the other hand the application must synchronize with the ISR because the ISR may be dispatched at any point. To ensure mutual exclusion we redefine the semantics of the monitor associated with an InterruptHandler object: acquisition of the monitor disables all interrupts of the same and lower priority; release of the monitor enables the interrupts again. This procedure ensures that the software thread cannot be interrupted by the interrupt handler when accessing shared data.

3.2.3 Using the Interrupt Handler. Figure 12 shows an example of an interrupt handler for the serial port receiver interrupt. The method handle() is the interrupt handler method and needs no synchronization as it cannot be interrupted by a software thread. However, the shared data needs to be declared volatile as it is changed by the device driver thread. Method read() is invoked by the device driver thread and the shared data is protected by the InterruptHandler monitor. The serial port interrupt handler uses the hardware object SerialPort to access the device.

3.2.4 Garbage Collection. When using the ISR handler approach it is not feasible to let interrupt handlers be paused during a lengthy stop-the-world collection. Using this GC strategy the entire heap is collected at once and it is not interleaved with execution. The collector can safely assume that data required to perform the collection will not change during the collection, and an interrupt handler shall not change data used by the GC to complete the collection. In the general case, this means that the interrupt handler is not allowed to create new objects, or change the graph of live objects.

With an incremental GC the heap is collected in small incremental steps. Write barriers in the mutator threads and nonpreemption sections in the GC thread synchronize the view of the object graph between the mutator threads and the GC thread. With incremental collection it is possible to allow object allocation and changing references inside an interrupt handler (as it is allowed in any normal thread). With a real-time GC the maximum blocking time due to GC synchronization with the mutator threads must be known.

Interruption of the GC during an object move can result in access to a stale copy of the object inside the handler. A solution to this problem is to allow for pinning of objects reachable by the handler (similar to immortal memory in the RTSJ). Concurrent collectors have to solve this issue for threads anyway. The simplest approach is to disable interrupt handling during the object copy. As this operation can be quite long for large arrays, several approaches to split the array into smaller chunks have been proposed [Bacon et al. 2003; Siebert 2002]. A Java processor may support incremental array copying with redirection of the access to the correct part of the array [Schoeberl and Puffitsch 2008]. Another solution is to abort the object copy when writing to the

```
public class SerialHandler extends InterruptHandler {

    // A hardware object represents the serial device
    private SerialPort sp;

    final static int BUF_SIZE = 32;
    private volatile byte buffer[];
    private volatile int wrPtr, rdPtr;

    public SerialHandler(SerialPort sp) {
        this.sp = sp;
        buffer = new byte[BUF_SIZE];
        wrPtr = rdPtr = 0;
    }

    // This method is scheduled by the hardware
    public void handle() {
        byte val = (byte) sp.data;
        // check for buffer full
        if ((wrPtr+1)%BUF_SIZE!=rdPtr) {
            buffer[wrPtr++] = val;
        }
        if (wrPtr>=BUF_SIZE) wrPtr=0;
        // enable interrupts again
        enableInterrupt();
    }

    // This method is invoked by the driver thread
    synchronized public int read() {
        if (rdPtr!=wrPtr) {
            int val = ((int) buffer[rdPtr++]) & 0xff;
            if (rdPtr>=BUF_SIZE) rdPtr=0;
            return val;
        } else {
            return -1;        // empty buffer
        }
    }
}
```

Fig. 12. An interrupt handler for a serial port receive interrupt.

object. It is also possible to use replication; during an incremental copy operation, writes are performed on both from-space and to-space object replicas, while reads are performed on the from-space replica.

3.3 Generic Configuration

An important issue for a HAL is a safe abstraction of device configurations. A definition of factories to create hardware and interrupt objects should be provided by board vendors. This configuration is isolated with the help of Java packages; only the objects and the factory methods are visible. The configuration abstraction is independent of the JVM. A device or interrupt can be represented by an identical hardware or interrupt object for different JVMs. Therefore, device drivers written in Java are JVM-independent.

3.3.1 Hardware Object Creation. The idea to represent each device by a single object or array is straightforward, the remaining question is: How are those objects created?

```
package com.board-vendor.io;

public class IOSystem {

    // some JVM mechanism to create the hardware objects
    private static ParallelPort pp = jvmPPCreate();
    private static SerialPort sp = jvmSPCreate(0);
    private static SerialPort gps = jvmSPCreate(1);

    public static ParallelPort getParallelPort() {
        return pp;
    }

    public static SerialPort getSerialPort() {..}
    public static SerialPort getGpsPort() {..}
}
```

Fig. 13. A factory with static methods for Singleton hardware objects.

An object that represents a device is a typical Singleton [Gamma et al. 1994]. Only a single object should map to one instance of a device. Therefore, hardware objects cannot be instantiated by a simple new: (1) they have to be mapped by some JVM mechanism to the device registers and (2) each device instance is represented by a single object.

Each device object is created by its own factory method. The collection of these methods is the board configuration, which itself is also a Singleton (the application runs on a single board). The Singleton property of the configuration is enforced by a class that contains only static methods. Figure 13 shows an example for such a class. The class IOSystem represents a system with three devices: a parallel port, as discussed before to interact with the environment, and two serial ports: one for program download and one which is an interface to a GPS receiver.

This approach is simple, but not very flexible. Consider a vendor who provides boards in slightly different configurations (e.g., with different number of serial ports). With the preceding approach each board requires a different (or additional) IOSystem class that lists all devices. A more elegant solution is proposed in the next section.

3.3.2 Board Configurations. Replacing the static factory methods by instance methods avoids code duplication; inheritance then gives configurations. With a factory object we represent the common subset of I/O devices by a base class and the variants as subclasses.

However, the factory object itself must still be a Singleton. Therefore the board-specific factory object is created at class initialization and is retrieved by a static method. Figure 14 shows an example of a base factory and a derived factory. Note how getBaseFactory() is used to get a single instance of the factory. We have applied the idea of a factory two times: the first factory generates an object that represents the board configuration. That object is itself a factory that generates the objects that interface to the hardware device.

The shown example base factory represents the minimum configuration with a single serial port for communication (mapped to System.in and System.out) represented by a SerialPort. The derived configuration ExtendedBoard contains an additional serial port for a GPS receiver and a parallel port for external control.

Furthermore, we show in Figure 14 a different way to incorporate the JVM mechanism in the factory: we define well-known constants (the memory addresses of the

```
public class BaseBoard {

    private final static int SERIAL_ADDRESS = ...;
    private SerialPort serial;
    BaseBoard() {
        serial = (SerialPort) jvmHWOCreate(SERIAL_ADDRESS);
    };
    static BaseBoard single = new BaseBoard();
    public static BaseBoard getBaseFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return serial; }

    // here comes the JVM internal mechanism
    Object jvmHWOCreate(int address) {...}
}

public class ExtendedBoard extends BaseBoard {

    private final static int GPS_ADDRESS = ...;
    private final static int PARALLEL_ADDRESS = ...;
    private SerialPort gps;
    private ParallelPort parallel;
    ExtendedBoard() {
        gps = (SerialPort) jvmHWOCreate(GPS_ADDRESS);
        parallel = (ParallelPort) jvmHWOCreate(PARALLEL_ADDRESS);
    };
    static ExtendedBoard single = new ExtendedBoard();
    public static ExtendedBoard getExtendedFactory() {
        return single;
    }
    public SerialPort getGpsPort() { return gps; }
    public ParallelPort getParallelPort() { return parallel; }
}
```

Fig. 14. A base class of a hardware object factory and a factory subclass.

devices) in the factory and let the native function jvmHWOCreate() return the correct device type.

Figure 15 gives a summary example of hardware object classes and the corresponding factory classes as an UML class diagram. The figure shows that all device classes subclass the abstract class HardwareObject. Figure 15 represents the simple abstraction as it is seen by the user of hardware objects.

3.3.3 Interrupt Handler Registration. We provide a base interrupt handling API that can be used both for non-RTSJ and RTSJ interrupt handling. The base class that is extended by an interrupt handler is shown in Figure 16. The handle() method contains the device server code. Interrupt control operations that have to be invoked before serving the device (i.e., interrupt masking and acknowledging) and after serving the device (i.e., interrupt reenabling) are hidden in the run() method of the base InterruptHandler, which is invoked when the interrupt occurs.

The base implementation of InterruptHandler also provides methods for enabling and disabling a particular interrupt or all local CPU interrupts and a special monitor implementation for synchronization between an interrupt handler thread and an application thread. Moreover, it provides methods for non-RTSJ registering and deregistering the handler with the hardware interrupt source.

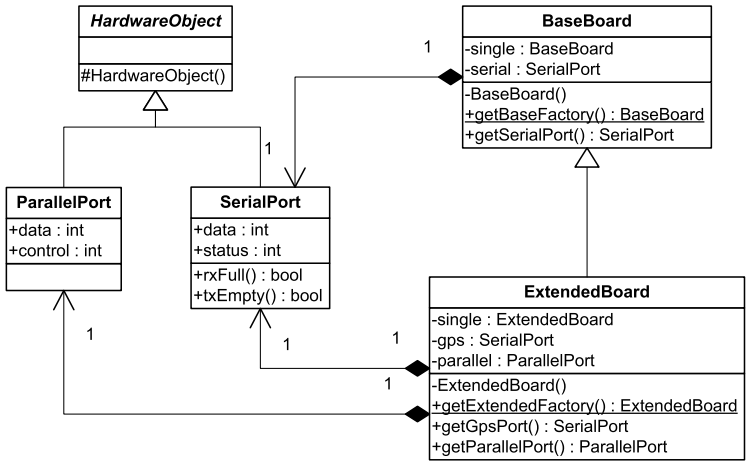

Fig. 15. Device object classes and board factory classes.

```
abstract public class InterruptHandler implements Runnable {
    ...

    public InterruptHandler(int index) { ... };

    protected void startInterrupt() { ... };
    protected void endInterrupt() { ... };

    protected void disableInterrupt() { ... };
    protected void enableInterrupt() { ... };
    protected void disableLocalCPUInterrupts() { ... };
    protected void enableLocalCPUInterrupts() { ... };

    public void register() { ... };
    public void unregister() { ... };

    abstract public void handle() { ... };

    public void run() {
        startInterrupt();
        handle();
        endInterrupt();
    }
}
```

Fig. 16. Base class for the interrupt handlers.

Registration of a RTSJ interrupt handler requires more steps (see Figure 17). The InterruptHandler instance serves as the RTSJ logic for a (bound) asynchronous event handler, which is added as handler to an asynchronous event which then is bound to the interrupt source.

3.4 Perspective

An interesting topic is to define a common standard for hardware objects and interrupt handlers for different platforms. If different device types (hardware chips) that do not share a common register layout are used for a similar function, the hardware objects

```
ih = new SerialInterruptHandler(); // logic of new BAEH

serialFirstLevelEvent = new AsyncEvent();
serialFirstLevelEvent.addHandler(
    new BoundAsyncEventHandler( null, null, null, null, null, false, ih )
    );

serialFirstLevelEvent.bindTo("INT4");
```

Fig. 17. Creation and registration of a RTSJ interrupt handler.

Table II. Embedded Java Architectures

	Direct (no OS)	Indirect (OS)
Interpreted	SimpleRTJ	Kaffe VM
Native	JOP	OVM

will be different. However, if the structure of the devices is similar, as is the case for the serial port on the three different platforms used for the implementation (see Section 4), the driver code that *uses* the hardware object is identical.

If the same chip (e.g., the 8250 type and compatible 16x50 devices found in all PCs for the serial port) is used in different platforms, the hardware object and the device driver, which also implements the interrupt handler, can be shared. The hardware object, the interrupt handler, and the visible API of the factory classes are independent of the JVM and the OS. Only the implementation of the factory methods is JVM-specific. Therefore, the JVM-independent HAL can be used to start the development of drivers for a Java OS on any JVM that supports the proposed HAL.

3.5 Summary

Hardware objects are easy to use for a programmer, and the corresponding definitions are comparatively easy to define for a hardware designer or manufacturer. For a standardized HAL architecture we proposed factory patterns. As shown, interrupt handlers are easy to use for a programmer who knows the underlying hardware paradigm, and the definitions are comparatively easy to develop for a hardware designer or manufacturer, for instance using the patterns outlined in this section. Hardware objects and interrupt handler infrastructure have a few subtle implementation points which are discussed in the next section.

4. IMPLEMENTATION

We have implemented the core concepts on four different JVMs[7] to validate the proposed Java HAL. Table II classifies the four execution environments according to two important properties: (1) whether they run on bare metal or on top of an OS and (2) whether Java code is interpreted or executed natively. Thereby we cover the whole implementation spectrum with our four implementations. Even though the suggested Java HAL is intended for systems running on bare metal, we include systems running on top of an OS because most existing JVMs still require an OS, and in order for them to migrate incrementally to run directly on the hardware they can benefit from supporting a Java HAL.

In the direct implementation a JVM without an OS is extended with I/O functionality. The indirect implementation represents an abstraction mismatch; we actually

[7]On JOP the implementation of the Java HAL is already in use in production code.

```
public final class SerialPort extends HardwareObject {
    // LSR (Line Status Register)
    public volatile int status;
    // Data register
    public volatile int data;
    ...
}
```

Fig. 18. A simple hardware object.

remap the concepts. Related to Figure 6 in the Introduction, OVM and Kaffe represent configuration (a), SimpleRTJ configuration (b), and JOP configuration (c).

The SimpleRTJ JVM [RTJ Computing 2000] is a small, interpreting JVM that does not require an OS. JOP [Schoeberl 2005, 2008] is a Java processor executing Java bytecodes directly in hardware. Kaffe JVM [Wilkinson 1996] is a complete, full-featured JVM supporting both interpretation and JIT compilation; in our experiments with Kaffe we have used interpretative execution only. The OVM JVM [Armbruster et al. 2007] is an execution environment for Java that supports compilation of Java bytecodes into the C language, and via a C compiler into native machine instructions for the target hardware. Hardware objects have also been implemented in the research JVM, CACAO [Krall and Grafl 1997; Schoeberl et al. 2008].

In the following we provide the different implementation approaches that are necessary for the very different JVMs. Implementing hardware objects was straightforward for most JVMs; it took about one day to implement them in JOP. In Kaffe, after familiarizing us with the structure of the JVM, it took about half a day of pair programming.

Interrupt handling in Java is straightforward in a JVM not running on top of an OS (JOP and SimpleRTJ). Kaffe and OVM both run under vanilla Linux or the real-time version Xenomai Linux [Xenomai Developers 2008]. Both versions use a distinct user/kernel mode and it is not possible to register a user-level method as interrupt handler. Therefore, we used threads at different levels to simulate the Java handler approach. The result is that the actual Java handler is the third- or even fourth-level handler. This solution introduces quite a lot of overheads due to the many context switches. However, it is intended to provide a stepping stone to allow device drivers in Java; the goal is a real-time JVM that runs on the bare hardware.

In this section we provide more implementation details than usual to help other JVM developers to add a HAL to their JVM. The techniques used for the JVMs can probably not be used directly. However, the solutions (or sometimes work-arounds) presented here should give enough insight to guide other JVM developers.

4.1 SimpleRTJ

The SimpleRTJ JVM is a small, simple, and portable JVM. We have ported it to run on the bare metal of a small 16-bit microcontroller. We have successfully implemented the support for hardware objects in the SimpleRTJ JVM. For interrupt handling we use the ISR handler approach described in Section 3.2. Adding support for hardware objects was straightforward, but adding support for interrupt handling required more work.

4.1.1 Hardware Objects. Given an instance of a hardware object as shown in Figure 18 one must calculate the base address of the I/O port range, the offset to the actual I/O port, and the width of the port at runtime. We have chosen to store the base address of the I/O port range in a field in the common superclass for all hardware objects (HardwareObject). The hardware object factory passes the platform- and device-specific

```
SerialPort createSerialPort(int baseAddress ) {
    SerialPort sp = new SerialPort(baseAddress);
    return sp;
}
```

Fig. 19. Creating a simple hardware object.

base address to the constructor when creating instances of hardware objects (see Figure 19).

In the put/getfield bytecodes the base address is retrieved from the object instance. The I/O port offset is calculated from the offset of the field being accessed: in the example in Figure 18 status has an offset of 0 whereas data has an offset of 4. The width of the field being accessed is the same as the width of the field type. Using these values the SimpleRTJ JVM is able to access the device register for either read or write.

4.1.2 Interrupt Handler. The SimpleRTJ JVM uses a simple stop-the-world garbage collection scheme. This means that within handlers, we prohibit use of the new keyword and writing references to the heap. These restrictions can be enforced at runtime by throwing a preallocated exception or at class loading by an analysis of the handler method. Additionally we have turned off the compaction phase of the GC to avoid the problems with moving objects mentioned in Section 3.2.4.

The SimpleRTJ JVM implements thread scheduling within the JVM. This means that it had to be refactored to allow for reentering the JVM from inside the first-level interrupt handler. We got rid of all global state (all global variables) used by the JVM and instead allocate shared data on the C stack. For all parts of the JVM to still be able to access the shared data we pass around a single pointer to that data. In fact we start a new JVM for the interrupt handler with a temporary (small) Java heap and a temporary (small) Java stack. Currently we use 512 bytes for each of these items, which have proven sufficient for running nontrivial interrupt handlers so far.

The major part of the work was making the JVM reentrant. The effort will vary from one JVM implementation to another, but since global state is a bad idea in any case JVMs of high quality use very little global state. Using these changes we have experimented with handling the serial port receive interrupt.

4.2 JOP

JOP is a Java processor intended for hard real-time systems [Schoeberl 2005, 2008]. All architectural features have been carefully designed to be time-predictable with minimal impact on average case performance. We have implemented the proposed HAL in the JVM for JOP. No changes inside the JVM (the microcode in JOP) were necessary. Only the creation of the hardware objects needs a JOP-specific factory.

4.2.1 Hardware Objects. In JOP, objects and arrays are referenced through an indirection called *handle*. This indirection is a lightweight read barrier for the compacting real-time GC [Schoeberl 2006; Schoeberl and Vitek 2007]. All handles for objects in the heap are located in a distinct memory region, the handle area. Besides the indirection to the *real* object the handle contains auxiliary data, such as a reference to the class information, the array length, and GC-related data. Figure 20 shows an example with a small object that contains two fields and an integer array of length 4. The object and the array on the heap just contain the data and no additional hidden fields. This object layout greatly simplifies our object to device mapping. We just need a handle where the indirection points to the memory-mapped device registers instead of into the heap. This configuration is shown in the upper part of Figure 20. Note that we do not need

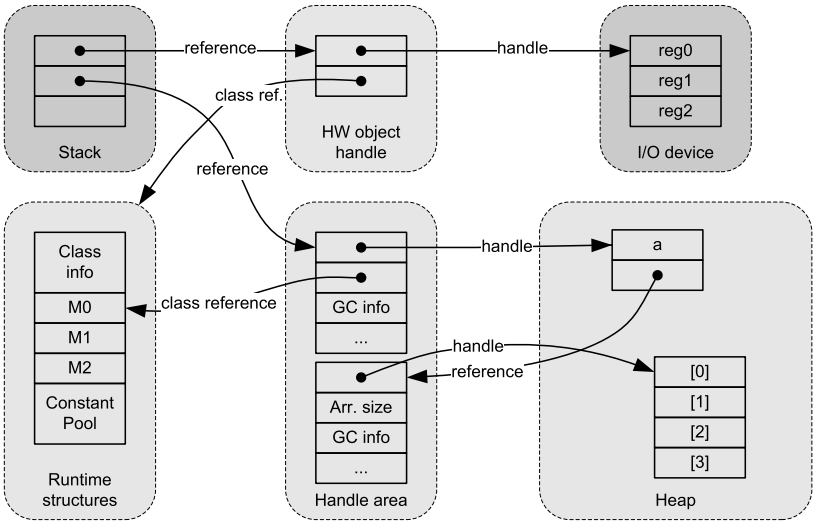

Fig. 20. Memory layout of the JOP JVM.

the GC information for the hardware object handles. The factory, which creates the hardware objects, implements this indirection.

As described in Section 3.3.1 we do not allow applications to create hardware objects; the constructor is private (or package visible). Figure 21 shows part of the hardware object factory that creates the hardware object SerialPort. Two static fields (SP_PTR and SP_MTAB) are used to store the handle to the serial port object. The first field is initialized with the base address of the I/O device; the second field contains a pointer to the class information.[8] The address of the static field SP_PTR is returned as the reference to the serial port object.

The class reference for the hardware object is obtained by creating a normal instance of SerialPort with new on the heap and copying the pointer to the class information. To avoid using native methods in the factory class we delegate JVM internal work to a helper class in the JVM system package as shown in Figure 21. That helper method returns the address of the static field SP_PTR as reference to the hardware object. All methods in class Native, a JOP system class, are *native*[9] methods for low-level functions, the code we want to avoid in application code. Method toInt(Object o) defeats Java's type safety and returns a reference as an int. Method toObject(int addr) is the inverse function to map an address to a Java reference. Low-level memory access methods are used to manipulate the JVM data structures.

To disallow the creation with new in normal application code, the visibility is set to package. However, the package visibility of the hardware object constructor is a minor issue. To access private static fields of an arbitrary class from the system class we had to change the runtime class information: we added a pointer to the first static primitive field of that class. As addresses of static fields get resolved at class linking, no such reference was needed so far.

[8]In JOP's JVM the class reference is a pointer to the method table to speed-up the invoke instruction. Therefore, the name is XX_MTAB.

[9]There are no real native functions in JOP; bytecode is the native instruction set. The very few native methods in class Native are replaced by special, unused bytecodes during class linking.

```
package com.jopdesign.io;

public class BaseFactory {

    // static fields for the handle of the hardware object
    private static int SP_PTR;
    private static int SP_MTAB;

    private SerialPort sp;

    IOFactory() {
        sp = (SerialPort) makeHWObject(new SerialPort(), Const.IO_UART1_BASE, 0);
    };

    ...

    // That's the JOP version of the JVM mechanism
    private static Object makeHWObject(Object o, int address, int idx) {
        int cp = Native.rdIntMem(Const.RAM_CP);
        return JVMHelp.makeHWObject(o, address, idx, cp);
    }
}

package com.jopdesign.sys;

public class JVMHelp {

    public static Object makeHWObject(Object o, int address, int idx, int cp) {
        // usage of native methods is allowed here as
        // we are in the JVM system package
        int ref = Native.toInt(o);
        // fill in the handle in the two static fields
        // and return the address of the handle as a
        // Java object
        return Native.toObject(address);
    }
}
```

Fig. 21. Part of a factory and the helper method for the hardware object creation in the factory.

4.2.2 Interrupt Handler. The original JOP [Schoeberl 2005, 2008] was a very puristic hard real-time processor. There existed only one interrupt: the programmable timer interrupt as time is the primary source for hard real-time events. All I/O requests were handled by periodic threads that polled for pending input data or free output buffers. During the course of this research we have added an interrupt controller to JOP and the necessary software layers.

Interrupts and exact exceptions are considered the hard part in the implementation of a processor pipeline [Hennessy and Patterson 2002]. The pipeline has to be drained and the complete processor state saved. In JOP there is a translation stage between Java bytecodes and the JOP internal microcode [Schoeberl 2008]. On a pending interrupt (or exception generated by the hardware) we use this translation stage to insert a special bytecode in the instruction stream. This approach keeps the interrupt completely transparent to the core pipeline. The special bytecode that is unused by the JVM specification [Lindholm and Yellin 1999] is handled in JOP as any other bytecode: execute microcode, invoke a special method from a helper class, or execute

```
static Runnable ih[] = new Runnable[Const.NUM_INTERRUPTS];
static SysDevice sys = IOFactory.getFactory().getSysDevice();

static void interrupt() {

    ih[sys.intNr].run();
}
```

Fig. 22. Interrupt dispatch with the static interrupt() method in the JVM helper class.

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        InterruptHandler ih = new InterruptHandler();
        IOFactory fact = IOFactory.getFactory();
        // register the handler
        fact.registerInterruptHandler(1, ih);
        // enable interrupt 1
        fact.enableInterrupt(1);
        .....
    }

    public void run() {
        System.out.println("Interrupt fired!");
    }
}
```

Fig. 23. An example Java interrupt handler as Runnable.

Java bytecode from JVM.java. In our implementation we invoke the special method interrupt() from a JVM helper class.

The implemented Interrupt Controller (IC) is priority-based. The number of interrupt sources can be configured. Each interrupt can be triggered in software by a IC register write as well. There is one global interrupt enable and each interrupt line can be enabled or disabled locally. The interrupt is forwarded to the bytecode/microcode translation stage with the interrupt number. When accepted by this stage, the interrupt is acknowledged and the global enable flag cleared. This feature avoids immediate handling of an arriving higher-priority interrupt during the first part of the handler. The interrupts have to be enabled again by the handler at a *convenient* time. All interrupts are mapped to the same special bytecode. Therefore, we perform the dispatch of the correct handler in Java. On an interrupt the static method interrupt() from a system internal class gets invoked. The method reads the interrupt number and performs the dispatch to the registered Runnable as illustrated in Figure 22. Note how a hardware object of type SysDevice is used to read the interrupt number.

The timer interrupt, used for the real-time scheduler, is located at index 0. The scheduler is just a plain interrupt handler that gets registered at mission start at index 0. At system startup, the table of Runnables is initialized with dummy handlers. The application code provides the handler via a class that implements Runnable and registers that class for an interrupt number. We reuse the factory presented in Section 3.3.1. Figure 23 shows a simple example of an interrupt handler implemented in Java.

For interrupts that should be handled by an event handler under the control of the scheduler, the following steps need to be performed on JOP.

(1) Create a SwEvent with the correct priority that performs the second-level interrupt handler work.

(2) Create a short first-level interrupt handler as Runnable that invokes fire() of the corresponding software event handler.
(3) Register the first-level interrupt handler as shown in Figure 23 and start the real-time scheduler.

In Section 5 we evaluate the different latencies of first- and second-level interrupt handlers on JOP.

4.3 Kaffe

Kaffe is an open-source[10] implementation of the JVM which makes it possible to add support for hardware objects and interrupt handlers. Kaffe requires a fully fledged OS such as Linux to compile and run. Although ports of Kaffe exist on uCLinux we have not been able to find a bare metal version of Kaffe. Thus even though we managed to add support of hardware objects and interrupt handling to Kaffe, it still cannot be used without an OS.

4.3.1 Hardware Objects. Hardware objects have been implemented in the same manner as in the SimpleRTJ, described in Section 4.1.

4.3.2 Interrupt Handler. Since Kaffe runs under Linux we cannot directly support the ISR handler approach. Instead we used the ISR event approach in which a thread blocks waiting for the interrupt to occur. It turned out that the main implementation effort was spent in the signaling of an interrupt occurrence from the kernel space to the user space.

We wrote a special Linux kernel module in the form of a character device. Through proper invocations of ioctl() it is possible to let the module install a handler for an interrupt (e.g., the serial interrupt, normally on IRQ 7). Then the Kaffe VM can make a blocking call to read() on the proper device. Finally the installed kernel handler will release the user space application from the blocked call when an interrupt occurs.

Using this strategy we have performed nontrivial experiments implementing a full interrupt handler for the serial interrupt in Java. Still, the elaborate setup requiring a special-purpose kernel device is far from our ultimate goal of running a JVM on the bare metal. Nevertheless the experiment has given valuable experience with interrupt handlers and hardware objects at the Java language level.

4.4 OVM

OVM [Armbruster et al. 2007] is a research JVM allowing many configurations; it is primarily targeted at implementing a large subset of RTSJ while maintaining reasonable performance. OVM uses ahead of time compilation via the C language: it translates both application and VM bytecodes to C, including all classes that might be later loaded dynamically at runtime. The C code is then compiled by GCC.

4.4.1 Hardware Objects. To compile Java bytecode into a C program, the OVM's Java-to-C compiler internally converts the bytecode into an Intermediate Representation (IR) which is similar to the bytecode, but includes more codes. Transformations at the IR level are both optimizations and operations necessary for correct execution, such as insertion of null-pointer checks. The produced IR is then translated into C, allowing the C compiler to perform additional optimizations. Transformations at the IR level, which is similar to the bytecode, are also typical in other JVM implementations, such as Sun's HotSpot.

[10] http://www.kaffe.org/

Reading from the device register serial.data, saving the result to the stack

original bytecode	*stack content*		*modified bytecode*	*stack content*
	{serial}			{serial}
GETFIELD data	{io port address}		GETFIELD data	{io port address}
		$\Longrightarrow$	INB	{inval}

Writing a value on the stack into the device register serial.data

original bytecode	*stack content*		*modified bytecode*	*stack content*
	{serial}, {outval}			{serial}, {outval}
PUTFIELD data	*empty*	$\Longrightarrow$	SWAP	{outval}, {serial}
		$\Longrightarrow$	GETFIELD data	{outval}, {io port address}
		$\Longrightarrow$	OUTB	*empty*

Fig. 24. Translation of bytecode for access to regular fields into bytecode for access to I/O port registers.

We base our access to hardware objects on IR instruction transformations. We introduce two new instructions outb and inb for byte-wide access to I/O ports. Then we employ OVM's instruction rewriting framework to translate accesses to hardware object fields, putfield and getfield instructions, into sequences centered around outb and inb where appropriate. We did not implement word-wide or double-word-wide access modes supported by a x86 CPU. We discuss how this could be done at the end of this section.

To minimize changes to the OVM code we keep the memory layout of hardware objects as if they were ordinary objects, and store port addresses into the fields representing the respective hardware I/O ports. Explained with the example from Figure 18, the instruction rewriting algorithm proceeds as follows: SerialPort is a subclass of HardwareObject; hence it is a hardware object, and thus accesses to all its public volatile int fields, status and data, are translated to port accesses to I/O addresses stored in those fields.

The translation (Figure 24) is very simple. In case of reads we append our new inb instruction after the corresponding getfield instruction in the IR: getfield will store the I/O address on the stack and inb will replace it by a value read from this I/O address. In case of writes we replace the corresponding putfield instruction by a sequence of swap, getfield, and outb. The swap rotates the two top elements on stack, leaving the hardware object reference on top of the stack and the value to store to the I/O port below it, The getfield replaces the object reference by the corresponding I/O address, and outb writes the value to the I/O port.

The critical part of hardware object creation is to set I/O addresses into hardware object fields. Our approach allows a method to turn off the special handling of hardware objects. In a hardware object factory method accesses to hardware object fields are handled as if they were fields of regular objects; we simply store I/O addresses to the fields.

A method can turn off the special handling of hardware objects with a marker exception mechanism which is a natural solution within OVM. The method declares to throw a PragmaNoHWIORegistersAccess exception. This exception is neither thrown nor caught, but the OVM IR-level rewriter detects the declaration and disables

rewriting accordingly. As the exception extends RuntimeException, it does not need to be declared in interfaces or in code calling factory methods. In Java 1.5, not supported by OVM, a standard substitute to the marker exception would be method annotation.

Our solution depends on the representation of byte-wide registers by 16-bit fields to hold the I/O address. However, it could still be extended to support multiple-width accesses to I/O ports (byte, 16-bit, and 32-bit) as follows: 32-bit I/O registers are represented by Java long fields, 16-bit I/O registers by Java int fields, and byte-wide I/O registers by Java short fields. The correct access width will be chosen by the IR rewriter based on the field type.

4.4.2 Interrupt Handler. Low-level support depends heavily on scheduling and preemption. For our experiments we chose the uniprocessor x86 OVM configuration with green threads running as a single Linux process. The green threads, delayed I/O operations, and handlers of asynchronous events, such as POSIX signals, are only scheduled at well-defined points (*pollchecks*) which are by default at back-branches at bytecode level and indirectly at Java-level blocking calls (I/O operations, synchronization calls, etc). When no thread is ready to run, the OVM scheduler waits for events using the POSIX select call.

As OS we use Xenomai RT Linux [Gerum 2004; Xenomai Developers 2008]. Xenomai tasks, which are in fact user-space Linux threads, can run either in the Xenomai primary domain or in the Xenomai secondary domain. In the primary domain they are scheduled by the Xenomai scheduler, isolated from the Linux kernel. In the secondary domain Xenomai tasks behave as regular real-time Linux threads. Tasks can switch to the primary domain at any time, but are automatically switched back to the secondary domain whenever they invoke a Linux system call. A single Linux process can have threads of different types: regular Linux threads, Xenomai primary domain tasks, and Xenomai secondary domain tasks. Primary domain tasks can wait on hardware interrupts with a higher priority than the Linux kernel. The Xenomai API provides the interrupts using the ISR event handler approach and supports *virtualization* of basic interrupt operations – disabling and enabling a particular interrupt or all local CPU interrupts. These operations have the same semantics as real interrupts, and disabling/enabling a particular one leads to the corresponding operation being performed at the hardware level.

Before our extension, OVM ran as a single Linux process with a single (native Linux) thread, a *main OVM thread*. This native thread implemented Java green threads. To support interrupts we add additional threads to the OVM process: for each interrupt source handled in OVM we dynamically add an interrupt listener thread running in the Xenomai primary domain. The mechanism that leads to invocation of the Java interrupt handler thread is illustrated in Figure 25.

Upon receiving an interrupt, the listener thread marks the pending interrupt in a data structure shared with the main OVM thread. When it reaches a pollcheck, it discovers that an interrupt is pending. The scheduler then immediately wakes-up and schedules the Java green thread that is waiting for the interrupt (IRQ server thread in the figure). To simulate the first-level ISR handler approach, this green thread invokes some handler method. In a non-RTSJ scenario the green thread invokes the run() method of the associated InterruptHandler (see Figure 16). In an RTSJ scenario (not shown in Figure 25), a specialized thread fires an asynchronous event bound to the particular interrupt source. It invokes the fire() method of the respective RTSJ's AsyncEvent. As mentioned in Section 3.3.3 the RTSJ logic of AsyncEventHandler (AEH) registered to this event should be an instance of InterruptHandler in order to allow the interrupt handling code to access basic interrupt handling operations.

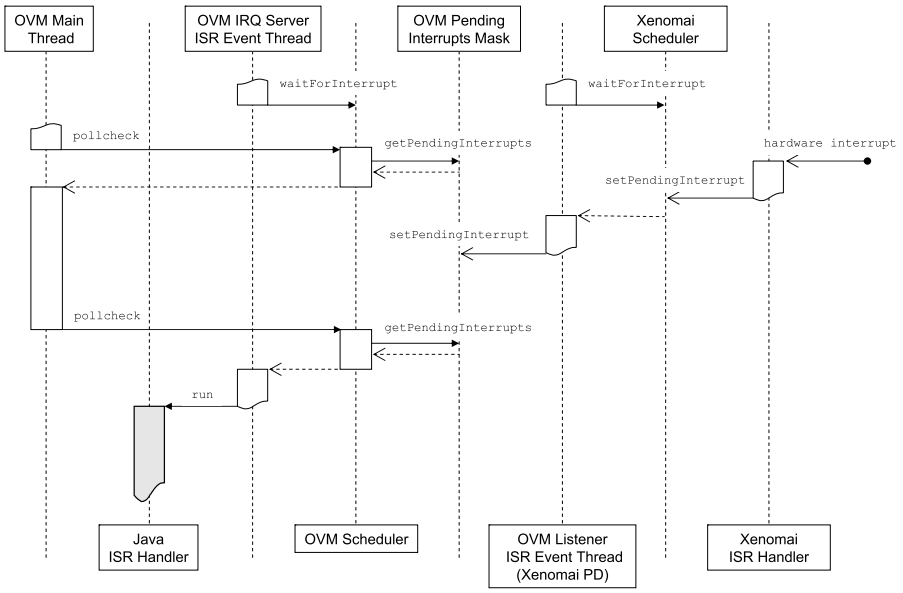

Fig. 25. Invocation of a Java interrupt handler under OVM/Xenomai.

As just explained, our first-level InterruptHandlers virtualize the interrupt handling operations for interrupt enabling, disabling, etc. Therefore, we have two levels of interrupt virtualization, one is provided by Xenomai to our listener thread, and the other one, on top of the first one, is provided by the OVM runtime to the InterruptHandler instance. In particular, disabling/enabling of local CPU interrupts is emulated, hardware interrupts are disabled/enabled, and interrupt completion is performed at the interrupt controller level (via the Xenomai API), and interrupt start is emulated; it only tells the listener thread that the interrupt was received.

The RTSJ scheduling features (deadline checking, interarrival time checking, delaying of sporadic events) related to release of the AEH should not require any further adaptations for interrupt handling. We could not test these features as OVM does not implement them.

OVM uses *thin monitors* which means that a monitor is only instantiated (*inflated*) when a thread has to block on acquiring it. This semantic does not match to what we need: disable the interrupt when the monitor is acquired to prevent the handler from interrupting. Our solution provides a special implementation of a monitor for interrupt handlers and inflate it in the constructor of InterruptHandler. This way we do not have to modify the monitorenter and monitorexit instructions and we do not slow down regular thin monitors (noninterrupt-based synchronization).

4.5 Summary

Support for hardware objects (see Section 3.1) and interrupt handling (see Section 3.2) to all four JVMs relies on common techniques. Accessing device registers through hardware objects extends the interpretation of the bytecodes putfield and getfield or redirects the pointer to the object. If these bytecodes are extended to identify the field being accessed as inside a hardware object, the implementation can use this information. Similarly, the implementation of interrupt handling requires changes to the bytecodes monitorenter and monitorexit or preinflating a specialized implementation of a Java monitor. In case of the bytecode extension, the extended codes specify if the

monitor being acquired belongs to an interrupt handler object. If so, the implementation of the actual monitor acquisition must be changed to disable/enable interrupts. Whether dealing with hardware or interrupt objects, we used the same approach of letting the hardware object and interrupt handler classes inherit from the superclasses HardwareObject and InterruptHandler respectively.

For JVMs that need a special treatment of bytecodes putfield and getfield (SimpleRTJ, Kaffe, and OVM) bytecode rewriting at runtime can be used to avoid the additional check of the object type. This is a standard approach (called *quick* bytecodes in the first JVM specification) in JVMs to speed-up field access of resolved classes.

Historically, registers of most x86 I/O devices are mapped to a dedicated I/O address space, which is accessed using dedicated instructions: port read and port writes. Fortunately, both the processor and Linux allow user-space applications running with administrator privileges to use these instructions and access the ports directly via iopl, inb, and outb calls. For both the Kaffe and OVM implementations we have implemented bytecode instructions putfield and getfield accessing hardware object fields by calls to iopl, inb, and outb.

Linux does not allow user-space applications to handle hardware interrupts. Only kernel space functionality is allowed to register interrupt handlers. We have overcome this issue in two different ways.

— For Kaffe we have written a special-purpose kernel module through which the user-space application (the Kaffe VM) can register interest in interrupts and get notified about interrupt occurrence.
— For OVM we have used the Xenomai real-time extension to Linux. Xenomai extends the Linux kernel to allow for the creation of real-time threads and allows user-space code to wait for interrupt occurrences.

Both these work-arounds allow an incremental transition of the JVMs and the related development libraries into a direct (bare metal) execution environment. In that case the work-arounds would no longer be needed.

If a compiling JVM is used (either as JIT or ahead-of-time) the compiler needs to be aware of the special treatment of hardware objects and monitors on interrupt handlers. One issue which we did not face in our implementations was the alignment of object fields. When device registers are represented by differently sized integer fields, the compiler needs to pack the data structure.

The restrictions within an interrupt handler are JVM-dependent. If an interruptible, real-time GC is used (as in OVM and JOP) objects can be allocated in the handler and the object graph may be changed. For a JVM with a stop-the-world GC (SimpleRTJ and Kaffe) allocations are not allowed because the handler can interrupt the GC.

5. EVALUATION AND CONCLUSION

Having implemented the Java HAL on four different JVMs we evaluate it on a several test applications, including a tiny Web server, and measure the performance of hardware accesses via hardware objects and the latency of Java interrupt handlers.

5.1 Qualitative Observations

For first tests we implemented a serial port driver with hardware objects and interrupt handlers. As the structure of the device registers is exactly the same on a PC, the platform for SimpleRTJ, and JOP, we were able to use the exact same definition of the hardware object SerialPort and the test programs on all four systems.

Using the serial device we run an embedded TCP/IP stack, implemented completely in Java, over a SLIP connection. The TCP/IP stack contains a tiny Web server and we

serve Web pages with a Java-only solution similar to the one shown in the Introduction in Figure 6. The TCP/IP stack, the tiny Web server, and the hardware object for the serial port are the same for all platforms. The only difference is in the hardware object creation with the platform-dependent factory implementations. The Web server uses hardware objects and polling to access the serial device.

5.1.1 A Serial Driver in Java. For testing the interrupt handling infrastructure in OVM we implemented a serial interrupt-based driver in Java and a demo application that sends back the data received through a serial interface. The driver part of the application is a full-duplex driver with support for hardware flow control and with detection of various error states reported by the hardware. The driver uses two circular buffers, one for receiving and the other for sending. The user part of the driver implements blocking getChar and putChar calls, which have (short) critical sections protected by the interrupt-disabling monitor. To reduce latencies the getChar call sets the DSR flag to immediately allow receiving more data and the putChar, after putting the character into the sending buffer, initiates immediately the sending, if this is not currently being done already by the interrupt machinery. The driver supports serial ports with a FIFO buffer. The user part of the demo application implements the loop-back using getChar and putChar. The user part is a RTSJ AsyncEventHandler which is fired when a new character is received. From a Java perspective this is a second-level interrupt handler, invoked after the corresponding serial event is fired from the first-level handler. To test the API described in the article we implemented two versions that differ in how the first-level handler is bound to the interrupt: (a) a RTSJ-style version where the first-level handler is also a RTSJ event handler bound using bindTo to the JVM provided first-level serial event, and (b) a non-RTSJ-style version where the first-level handler is registered using a InterruptHandler.register call. We have stress-tested the demo application and the underlying modified OVM infrastructure by sending large files to it through the serial interface and checked that they were returned intact.

5.1.2 The HAL in Daily Use. The original idea for hardware objects evolved during development of low-level software on the JOP platform. The abstraction with read and write functions and using constants to represent I/O addresses just *felt* wrong with Java. Currently hardware objects are used all over in different projects with JOP. Old code has been refactored to some extent, but new low-level code uses only hardware objects. By now low-level I/O is integrated into the language, for example, autocompletion in the Eclipse IDE makes it easy to access the factory methods and fields in the hardware object.

For experiments with an on-chip memory for thread-local scope caching [Wellings and Schoeberl 2009] in the context of a chip-multiprocessor version of JOP, the hardware array abstraction greatly simplified the task. The on-chip memory is mapped to a hardware array and the RTSJ-based scoped memory uses it. Creation of an object within this special scope is implemented in Java and is safe because the array bounds checks are performed by the JVM.

5.1.3 JNI vs. Hardware Objects. JNI provides a way to access the hardware without changing the code of the JVM. Nevertheless, with a lack of commonly agreed API, using it for each application would be redundant and error prone. It would also add dependencies to the application: hardware platform and the operating system (the C API for accessing the hardware is not standardized). The build process is complicated by adding C code to it as well. Moreover, the system needs to support shared libraries, which is not always the case for embedded operating systems (an example is RTEMS, used by ESA).

In addition, JNI is typically too heavyweight to implement trivial calls such as port or memory access efficiently (no GC interaction, no pointers, no threads interaction, no blocking). Even JVMs that implement JNI usually have some other internal lightweight native interface which is the natural choice for hardware access. This leads us back to a Java HAL as illustrated here.

5.1.4 OVM-Specific Experience. Before the addition of hardware objects, OVM did not allow hardware access because it did not and does not have JNI or any other native interface for user Java code. OVM has a simplified native interface for the virtual machine code which indeed we used when implementing the hardware objects. This native interface can as well be used to modify OVM to implement user-level access to hardware via regular method calls. We have done this to implement a benchmark to measure HWO/native overheads (later in this section). As far as simple port access is concerned, none of the solutions is strictly better from the point of the JVM: the bytecode manipulation to implement hardware objects was easy, as well as adding code to propagate native port I/O calls to user code. Thanks to ahead-of-time compilation and the simplicity of the native interface, the access overhead is the same.

The OVM compiler is fortunately not "too smart" so it does not get in the way of supporting hardware objects: if a field is declared volatile side-effects of reading of that field are not a problem for any part of the system.

The API for interrupt handling added to OVM allows full control over interrupts, typically available only to the operating system. The serial port test application has shown that, at least for a simple device; it really allows us to write a driver. An interesting feature of this configuration is that OVM runs in user space and therefore it greatly simplifies development and debugging of Java-only device drivers for embedded platforms.

5.2 Performance

Our main objective for hardware objects is a clean object-oriented interface to hardware devices. Performance of device register access is an important goal for relatively slow embedded processors; thus we focus on that in the following. It matters less on general-purpose processors where the slow I/O bus essentially limits the access time.

5.2.1 Measurement Methodology. Execution time measurement of single instructions is only possible on simple in-order pipelines when a cycle counter is available. On a modern superscalar architecture, where hundreds of instructions are in flight each clock cycle, direct execution time measurement becomes impossible. Therefore, we performed a bandwidth-based measurement. We measure how many I/O instructions per second can be executed in a tight loop. The benchmark program is self-adapting and increases the loop count exponentially until the measurement runs for more than one second and the iterations per second are reported. To compensate for the loop overhead we perform an overhead measurement of the loop and subtract that overhead from the I/O measurement. The I/O bandwidth b is obtained as follows.

$$b = \frac{cnt}{t_{test} - t_{ovhd}}$$

Figure 26 shows the measurement loop for the read operation in method test() and the overhead loop in method overhead(). In the comment above the method the bytecodes of the loop kernel are shown. We can see that the difference between the two loops is the single bytecode getfield that performs the read request.

5.2.2 Execution Time. In Table III we compare the access time with native functions to the access via hardware objects. The execution time is given in clock cycles. We

```
public class HwoRead extends BenchMark {

    SysDevice sys = IOFactory.getFactory().getSysDevice();

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ALOAD 2
    GETFIELD com/jopdesign/io/SysDevice.uscntTimer : I
    IADD
    ISTORE 3
*/
    public int test(int cnt) {

        SysDevice s = sys;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+s.uscntTimer;
        }
        return a;
    }

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ILOAD 2
    IADD
    ISTORE 3
*/
    public int overhead(int cnt) {

        int xxx = 456;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+xxx;
        }
        return a;
    }
}
```

Fig. 26. Benchmark for the read operation measurement.

scale the measured I/O bandwidth b with the clock frequency f of the system under test by $n = \frac{f}{b}$.

We have run the measurements on a 100 MHz version of JOP. As JOP is a simple pipeline, we can also measure short bytecode instruction sequences with the cycle counter. Those measurements provided the exact same values as the ones given by our benchmark, such that they validated our approach.

Table III. Access Time to a Device Register in Clock Cycles

	JOP		OVM		SimpleRTJ		Kaffe	
	read	write	read	write	read	write	read	write
native	5	6	5517	5393	2588	1123	11841	11511
HW Object	13	15	5506	5335	3956	3418	9571	9394

On JOP the native access is faster than using hardware objects because a native access is a special bytecode and not a native function call. The special bytecode accesses memory directly where the bytecodes putfield and getfield perform a null pointer check and indirection through the handle for the field access. Despite the slower I/O access via hardware objects on JOP, the access is fast enough for all currently available devices. Therefore, we will change all device drivers to use hardware objects. The measurement for OVM was run on a Dell Precision 380 (Intel Pentium 4, 3.8 GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux (Ubuntu 7.10, Linux 2.6.24.3 with Xenomai-RT patch). OVM was compiled without Xenomai support and the generated virtual machine was compiled with all optimizations enabled. As I/O port we used the printer port. Access to the I/O port via a hardware object is just slightly faster than access via native methods. This was expected as the slow I/O bus dominates the access time. On the SimpleRTJ JVM the native access is faster than access to hardware objects. The reason is that the JVM does not implement JNI, but has its own proprietary, more efficient way to invoke native methods. It is done in a prelinking phase where the invokestatic bytecode is instrumented with information to allow an immediate invocation of the target native function. On the other hand, using hardware objects needs a field lookup that is more time consuming than invoking a static method. With bytecode-level optimization at class load time it would be possible to avoid the expensive field lookup.

We measured the I/O performance with Kaffe on an Intel Core 2 Duo T7300, 2.00 GHz with Linux 2.6.24 (Fedora Core 8). We used access to the serial port for the measurement. On the interpreting Kaffe JVM we notice a difference between the native access and hardware object access. Hardware objects are around 20% faster.

5.2.3 Summary. For practical purposes the overhead on using hardware objects is insignificant. In some cases there may even be an improvement in performance. The benefits in terms of safe and structured code should make this a very attractive option for Java developers.

5.3 Interrupt Handler Latency

5.3.1 Latency on JOP. To measure interrupt latency on JOP we use a periodic thread and an interrupt handler. The periodic thread records the value of the cycle counter and triggers the interrupt. In the handler the counter is read again and the difference between the two is the measured interrupt latency. A plain interrupt handler as Runnable takes a constant 234 clock cycles (or 2.3 µs for a 100 MHz JOP system) between the interrupt occurrence and the execution of the first bytecode in the handler. This quite large time is the result of two method invocations for the interrupt handling: (1) invocation of the system method interrupt() and (2) invocation of the actual handler. For more time-critical interrupts the handler code can be integrated in the system method. In that case the latency drops down to 0.78 µs. For very low-latency interrupts the interrupt controller can be changed to emit different bytecodes depending on the interrupt number, then we avoid the dispatch in software and can implement the interrupt handler in microcode.

Table IV. Interrupt (and polling) Latencies in Microseconds

	Median (µs)	3rd Quartile (µs)	95% Quantile (µs)	Maximum (µs)
Polling	3	3	3	8
Kernel	14	16	16	21
Hard	14	16	16	21
User	17	19	19	24
Ovm	59	59	61	203

We have integrated the two-level interrupt handling at the application level. We set up two threads: one periodic thread, that triggers the interrupt, and a higher-priority event thread that acts as second-level interrupt handler and performs the handler work. The first-level handler just invokes fire() for this second-level handler and returns. The second-level handler gets scheduled according to the priority. With this setup the interrupt handling latency is 33 µs. We verified this time by measuring the time between fire of the software event and the execution of the first instruction in the handler directly from the periodic thread. This took 29 µs and is the overhead due to the scheduler. The value is consistent with the measurements in Schoeberl and Vitek [2007]. There we measured a minimum useful period of 50 µs for a high-priority periodic task.

The runtime environment of JOP contains a concurrent real-time GC [Schoeberl and Vitek 2007]. The GC can be interrupted at a very fine granularity. During sections that are not preemptive (data structure manipulation for a new and write-barriers on a reference field write) interrupts are simply turned off. The copy of objects and arrays during the compaction phase can be interrupted by a thread or interrupt handler [Schoeberl and Puffitsch 2008]. Therefore, the maximum blocking time is in the atomic section of the thread scheduler and not in the GC.

5.3.2 Latency on OVM/Xenomai. For measuring OVM/Xenomai interrupt latencies, we have extended an existing interrupt latency benchmark, written by Jan Kiszka from the Xenomai team [Xenomai Developers 2008]. The benchmark uses two machines connected over a serial line. The *log* machine, running a regular Linux kernel, toggles the RTS state of the serial line and measures the time it takes for the *target* machine to toggle it back.

To minimize measuring overhead the *log* machine uses only polling and disables local CPU interrupts while measuring. Individual measurements are stored in memory and dumped at shutdown so that they can be analyzed offline. We have made 400,000 measurements in each experiment, reporting only the last 100,000 (this was to warm-up the benchmark, including memory storage for the results). The *log* machine toggles the RTS state regularly with a given period.

We have tested 5 versions of the benchmark on the *target* machine: a polling version written in C (*polling*), a kernel-space interrupt handler in C/Xenomai running out of control of the Linux scheduler (*kernel*), a hard-realtime kernel-space interrupt handler running out of control of both the Xenomai scheduler and the Linux scheduler (*hard*), a user-space interrupt handler written in C/Xenomai (*user*), and finally an interrupt handler written in Java/OVM/Xenomai (*ovm*).

The results are shown in Table IV. The median latency is 3 µs for polling, 14 µs for both kernel-space handlers (hard and kernel), 17 µs for user-space C handler (user), and 59 µs for Java handler in OVM (ovm). Note that the table shows that the overhead of using interrupts over polling is larger than the overhead of handling interrupts in user space over kernel space. The maximum latency of OVM was 203 µs, due to

infrequent pauses. Their frequency is so low that the measured 95% quantile is only 61 µs.

The experiment was run on Dell Precision 380 (Intel Pentium 4 3.8 GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux (Ubuntu 7.10, Linux 2.6.24.3 with Xenomai-RT patch). As Xenomai is still under active development we had to use Xenomai work-arounds and bugfixes, mostly provided by Xenomai developers, to make OVM on Xenomai work.

5.3.3 Summary. The overhead for implementing interrupt handlers is very acceptable since interrupts are used to signal relatively infrequently occurring events like end of transmission, loss of carrier, etc. With a reasonable work division between first-level and second-level handlers, the proposal does not introduce dramatic blocking terms in a real-time schedulability analysis, and thus it is suitable for embedded systems.

5.4 Discussion

5.4.1 Safety Aspects. Hardware objects map object fields to the device registers. When the class that represents a device is correct, access to it is safe; it is not possible to read from or write to an arbitrary memory address. A memory area represented by an array is protected by Java's array bounds check.

5.4.2 Portability. It is obvious that hardware objects are platform-dependent; after all the idea is to have an interface to the bare metal. Nevertheless, hardware objects give device manufacturers an opportunity to supply supporting factory implementations that fit into Java's object-oriented framework and thus cater for developers of embedded software. If the same device is used on different platforms, the hardware object is portable. Therefore, standard hardware objects can evolve.

5.4.3 Compatibility with the RTSJ Standard. As shown for the OVM implementation, the proposed HAL is compatible with the RTSJ standard. We consider it to be a very important point since many existing systems have been developed using such platforms or subsets thereof. In further development of such applications existing and future interfacing to devices may be refactored using the proposed HAL. It will make the code safer and more structured and may assist in possible ports to new platforms.

5.5 Perspective

The many examples in the text show that we achieved a representation of the hardware close to being platform-independent. Also, they show that it is possible to implement system-level functionality in Java. As future work we consider to add devices drivers for common devices such as network interfaces[11] and hard disc controllers. On top of these drivers we will implement a file system and other typical OS-related services towards our final goal of a Java-only system.

An interesting question is whether a common set of *standard* hardware objects is definable. The SerialPort was a lucky example. Although the internals of the JVMs and the hardware were different one compatible hardware object worked on all platforms. It should be feasible that a chip manufacturer provides, beside the data sheet that describes the registers, a Java class for the register definitions of that chip. This definition can be reused in all systems that use that chip, independent of the JVM or OS.

Another interesting idea is to define the interaction between the GC and hardware objects. We stated that the GC should not collect hardware objects. If we relax this

[11]A device driver for a CS8900-based network chip is already part of the Java TCP/IP stack.

restriction we can redefine the semantics of collecting an object: on running the finalizer for a hardware object the device can be put into sleep mode.

ACKNOWLEDGMENTS

We wish to thank Andy Wellings for his insightful comments on an earlier version of the article. We also thank the reviewers for their detailed comments that helped to enhance the original submission.

REFERENCES

aJILE. 2000. aj-100 real-time low power Java processor. Preliminary data sheet.

ARMBRUSTER, A., BAKER, J., CUNEI, A., FLACK, C., HOLMES, D., PIZLO, F., PLA, E., PROCHAZKA, M., AND VITEK, J. 2007. A real-time Java virtual machine with applications in avionics. *Trans. Embed. Comput. Sys. 7,* 1, 1–49.

BACON, D. F., CHENG, P., AND RAJAN, V. T. 2003. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, New York, 285–298.

BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., AND TURNBULL, M. 2000. *The Real-Time Specification for Java*. Java Series. Addison-Wesley.

BURNS, A. AND WELLINGS, A. J. 2001. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX* 3rd Ed. Addison-Wesley Longman Publishing.

CASKA, J. accessed 2009. micro [μ] virtual-machine. `http://muvium.com/`.

CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. 2001. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev. 35,* 5, 73–88.

FELSER, M., GOLM, M., WAWERSICH, C., AND KLEINÖDER, J. 2002. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*. 45–58.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional.

GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. ACM Press, New York, 1–11.

GERUM, P. 2004. Xenomai - Implementing a RTOS emulation framework on GNU/Linux. `http://www.xenomai.org/documentation/branches/v2.4.x/pdf/xenomai.pdf`.

GROUP, T. C. 2008. Trusted computing. `https://www.trustedcomputinggroup.org/`.

HANSEN, P. B. 1977. *The Architecture of Concurrent Programs*. Prentice-Hall Series in Automatic Computing. Prentice-Hall.

HENNESSY, J. AND PATTERSON, D. 2002. *Computer Architecture: A Quantitative Approach* 3rd Ed. Morgan Kaufmann Publishers, Palo Alto, CA.

HENTIES, T., HUNT, J. J., LOCKE, D., NILSEN, K., SCHOEBERL, M., AND VITEK, J. 2009. Java for safety-critical applications. In *Proceedings of the 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'09)*.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D. E., AND PISTER, K. S. J. 2000. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. 93–104.

HUNT, G., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. D. 2005. An overview of the singularity project. Tech. rep. MSR-TR-2005-135, Microsoft Research (MSR).

KORSHOLM, S., SCHOEBERL, M., AND RAVN, A. P. 2008. Interrupt handlers in Java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'08)*. IEEE Computer Society.

KRALL, A. AND GRAFL, R. 1997. CACAO – A 64 bit JavaVM just-in-time compiler. In *Proceedings of the Workshop on Java for Science and Engineering Computation (PPoPP'97)*, G. C. Fox and W. Li Eds. ACM.

KREUZINGER, J., BRINKSCHULTE, U., PFEFFER, M., UHRIG, S., AND UNGERER, T. 2003. Real-Time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocess. Microsyst. 27,* 1, 19–31.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification* 2nd Ed. Addison-Wesley, Reading, MA.

LOHMEIER, S. 2005. Jini on the Jnode Java os. `http://monochromata.de/jnodejini.html`.

PHIPPS, G. 1999. Comparing observed bug and productivity rates for java and c++. *Softw. Pract. Exper. 29,* 4, 345–358.

RAVN, A. P. 1980. Device monitors. *IEEE Trans. Softw. Engin. 6,* 1, 49–53.

RTJ COMPUTING. 2000. SimpleRTJ a small footprint Java VM for embedded and consumer devices. `http://www.rtjcom.com/`.

SCHOEBERL, M. 2005. Jop: A java optimized processor for embedded real-time systems. Ph.D. thesis, Vienna University of Technology.

SCHOEBERL, M. 2006. Real-Time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. IEEE, 424–432.

SCHOEBERL, M. 2008. A Java processor architecture for embedded real-time systems. *J. Syst. Archit. 54/1–2*, 265–286.

SCHOEBERL, M. AND VITEK, J. 2007. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'07)*. ACM Press, 85–93.

SCHOEBERL, M. AND PUFFITSCH, W. 2008. Non-Blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'08)*. ACM Press.

SCHOEBERL, M., KORSHOLM, S., THALINGER, C., AND RAVN, A. P. 2008. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'08)*. IEEE Computer Society.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39,* 9, 1175–1185.

SIEBERT, F. 2002. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books.

SIMON, D., CIFUENTES, C., CLEAL, D., DANIELS, J., AND WHITE, D. 2006. Java on the bare metal of wireless sensor devices: The squawk Java virtual machine. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06)*. ACM Press, New York, 78–88.

WELLINGS, A. AND SCHOEBERL, M. 2009. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'09)*. IEEE Computer Society.

WILKINSON, T. 1996. Kaffe – A virtual machine to run java code. http://www.kaffe.org.

WIRTH, N. 1977. Design and implementation of modula. *Softw. Pract. Exper. 7*, 3–84.

WIRTH, N. 1982. *Programming in Modula-2*. Springer.

XENOMAI DEVELOPERS. 2008. Xenomai: Real-Time framework for Linux. `http://www.xenomai.org`.

Received August 2008; revised July 2009; accepted February 2010