Bringing EPOS components to embedded Java

Mateus Krepsky Ludwich and Antônio Augusto Fröhlich

Laboratory for Software and Hardware Integration – LISHA
Federal University of Santa Catarina – UFSC
P.O.Box 476, 880400900 - Florianópolis - SC - Brazil
{mateus,guto}@lisha.ufsc.br
http://www.lisha.ufsc.br

Abstract. Java implementations that focus on embedded systems must provide a way to interact with hardware devices such as sensors, actuators, transmitters, receivers, and timers in an abstract way. Besides that, the application and runtime must have a small footprint since embedded systems usually have memory constraints.
In this paper we show a way to interface hardware components with embedded Java applications. This interfacing is achieved using the foreign function interface of KESO Java Virtual Machine that does the binding between Java methods and C functions at compile-time. Using EPOS to support KESO execution, we provide a portable and economical way to run Java applications. We achieved a total footprint of less then 33KB, including the application and runtime support.

Keywords: Java, Embedded Systems, Foreign Function Interface

1 Introduction

Java implementations that have embedded systems as targets must provide developers with a way to control hardware devices. This is needed since embedded system applications run close to the hardware in the sense that they use hardware devices such as sensors and actuators to interact with the environment, transmitters and receivers for communication, and timers for real time operations. At the same time, Java applications for embedded systems must have a footprint according to the memory constraints imposed by many embedded systems.

Foreign Function Interface (FFI) like the *Java Native Interface* is the mechanism used by Java platforms to access hardware devices and memory. In fact, several Java packages such as java.io, java.net and java.awt are implemented using FFI facilities [15]. However, as we will explain in section 2, the main FFIs provided by Java have limitations to deal with embedded systems, because they are too onerous or because of design limitations.

This work demonstrates how we interface hardware components to Java applications. The Embedded Parallel Operating System (EPOS) abstracts hardware devices as components and the Foreign Function Interface of KESO Java

Virtual Machine (JVM) was used to creating the binding between these components and their Java counterparts.

EPOS *hardware mediators* sustain an interface contract between system abstractions (e.g. threads) and the machine allowing for these abstractions machine-independence. [19]. Since there is a mediator for each hardware device being abstracted, providing Java with these mediators allows a fine-grained control of hardware devices.

KESO JVM focuses on embedded systems and uses an ahead-of-time compilation strategy to translate programs from Java bytecode to C. The KESO FFI also uses a static approach generating the specified C code to native methods.

The next sections of this paper are organized in this way: section 2 presents an overview of the main requirements that Java must fulfill in the embedded systems scenario, showing approaches to deal with these requirements. We exemplify each approach using Java solutions from related work and using the KESO JVM. Section 3 presents KESO Java Virtual Machine and its main concepts, including KESO FFI. We present our approach to interface hardware components with Java applications in section 4, and we present a case study in section 5. Our final considerations are presented in section 6.

2 Java requirements for embedded systems

The requirements that Java must fulfill in the embedded systems scenario can be grouped in three categories: direct hardware access, elimination of java bytecode interpreter overhead, and runtime support and class libraries handling. This section explain these requirements categories as the approaches to deal with them.

2.1 Direct hardware access

The Java programming language does not provides the concept of *pointer* like C and C++ does. The address of *reference variables*, used to access Java objects, is just known by the Java Virtual Machine that handles all memory accesses. As major hardware devices are mapped in memory addresses, direct access to them is a problem to Java language. Foreign Function Interface (FFI) is the approach used by Java to overcome this limitation.

Foreign Function Interface (FFI) is a mechanism that allows programs written in a certain programming language to use constructions of programs written in another. FFIs are used by Java platforms to provide direct access to memory and I/O devices. FFIs can also be employed to reuse code written in other programming languages such as C and C++ and to embed JVMs into native applications allowing them to access Java functionality ([15], [14]).

Java Native Interface (JNI) is the main Java FFI, which is used in *Java Standard Edition* platform (JSE) [15]. In JNI, the binding of native code is performed during runtime. The Java *class loaders* are used to search for the native method and to add them into JVM. The same class loaders load Java class files

dynamically into JVM. Class loaders increase the need for runtime memory and increase the JVM size. Because of that, they are avoided in embedded systems.

The *Java Micro Edition* (JME) platform uses a lightweight FFI, called *K Native Interface* KNI [24]. KNI does not use class loaders, avoiding the memory overhead of JNI. In KNI the binding between Java and native code is performed statically, during compile time. However the design of KNI imposes some limitations. KNI forbids creating new Java objects (other than strings), and forbids calling Java methods from native code. Besides that, in KNI the only native methods that can be invoked are the ones pre-built into the JVM. There is no Java-level API to invoke others native methods. By consequence, it is difficult to create new hardware drivers through KNI.

KESO FFI, presented in section 3, focuses on embedded systems. Like KNI, KESO FFI does not use class loaders. But unlike KNI, KESO FFI provides for the programmer a Java-level API to create new native code bindings. Also there is no problem of native code calling Java code since KESO and KESO FFI generate C code.

2.2 Bytecode interpreter overhead

An overhead that embedded systems would like to avoid is the one of the Java bytecode interpreter. The Java bytecode interpreter causes a memory overhead because it is aways together with the applications. It also causes an execution time overhead since bytecode need to be interpreted by software and them executed. Approaches to deal with this problem are *Just-in-time* compilation, *Java processors*, and *Ahead-of-time* compilation.

Translations of Java bytecode into native code during execution time, known as *Just-in-time* compilation, is a widespread technique used by Java Standard Edition JVM. Gal et al. demonstrate the possibility of using JIT compilation for resource-constrained devices also [10]. However *Just-in-time* is just a partial solution to bytecode interpreter overhead problem. It solves part of the execution time overhead, since methods are interpreted once and then translated into native code. But the memory overhead remains.

Another method in which to eliminate the runtime overhead of Java bytecode interpreters is building the interpreter directly onto the hardware. In this case Java bytecode is the "native code" of such machines. This approach is implemented by the so-called *Java Processors*, of which Picojava I and II, Jazelle and JOP are examples [16], [20], [4], [21]. Java processors are not a practical solution since major embedded systems are implemented using more generic propose processors.

A more widespread used solution to eliminate the runtime overhead of Java bytecode interpreters is to translate source code or Java bytecode into a lower level language before program execution. This approach is called *Ahead-of-time compilation* ([3], [5], [17]). The *KESO compiler*, presented in section 3, is also an ahead-of-time compiler that translates a program written in Java bytecode to C.

2.3 Runtime support and class libraries

Even interpreting bytecode or compiling it into native code, the necessity of some runtime support still remains, for example the garbage collector. This runtime can lie in a preexistent library or can be generated according to the needs of the application. The same is valid for the classes that a Java application will need.

In [13] a framework is presented for the development and deployment of Java applications for embedded system devices which uses a runtime environment generated from specifications of the hardware and software platform which will be used by the application.

Secchi [22] proposes a tool for bytecode analysis and generation of application-specific class libraries. Pereira [18] proposes the use of these kinds of tools to build an application-oriented JVM.

KESO builder, presented in section 3, analyzes the application bytecode, creating a call graph, that is used to determine with parts of JVM is need by the application. Then is generated only the necessary code for providing runtime support to the application. The same bytecode analysis is used to eliminate classes, methods, and fields not accessed by the application.

3 KESO java virtual machine

KESO is a multi-Java Virtual Machine (JVM) that focuses on embedded devices and networks of micro controllers [25]. KESO relies on the OSEK/VDX operating system [8]. OSEK/VDX organization specifies standards for automotive systems, including the operating system standard used by KESO. Because KESO is based on OSEK, it exports the concepts of this operating system to Java applications, including statically allocated tasks, scheduling policies, interrupt and event mechanism, and communication [12]. Figure 1, taken from [23], shows KESO architecture and its main concepts.

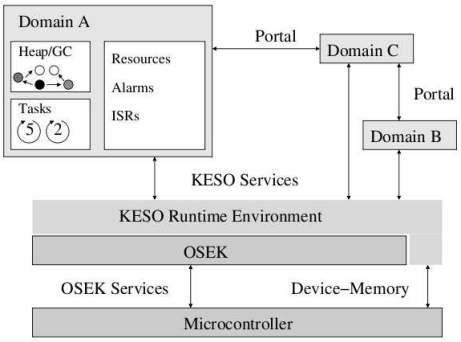

Fig. 1. KESO architecture [23].

Tasks are the schedulable unit on OSEK systems. They have fixed priority and are allocated statically during the system generation. KESO uses the concept of OSEK Tasks as a substitute for Java Threads. It is well suited since KESO focuses on static embedded systems, which allocate resources that will be used by the system application during design time.

Although not explored in this work, KESO can be used as a distributed system composed by *domains*. A domain appears to the application developer as a self-contained JVM which has its own heap and static class fields. KESO is called *multi-JVM* just because of that. A task in KESO just belongs to one domain and, like all other types of objects, cannot cross domain boundaries. The concept of domain was first introduced by JX [11] and is a way in which to perform memory protection in software that can be used when the target platform is absent of the memory protection unit (MPU) and memory management unit (MMU).

Inter-domain communication is performed using *portals*. A domain can provide a *portal service* that consists of a Java interface that offers *services* to other domains. A task from another domain can import the offered service using a global name service.

Although KESO is a JVM in the sense that it runs Java programs, it does not have byte code interpretation overhead. This happens because the *KESO Builder* translates Java bytecode into C code in an ahead-of-time fashion (i.e. before the program execution). Figure 2, taken from [25], shows this process. Java application source files and KESO class library are compiled to byte code using a standard Java compiler. Then, KESO Builder translates the class files into C source files and also generates the C source files corresponding to JVM features that will be used by the application (e.g. garbage collector). During the process, KESO Builder analyzes the bytecode and eliminates classes, methods, and fields not accessed by the application and eliminates, when possible, virtual method calls.

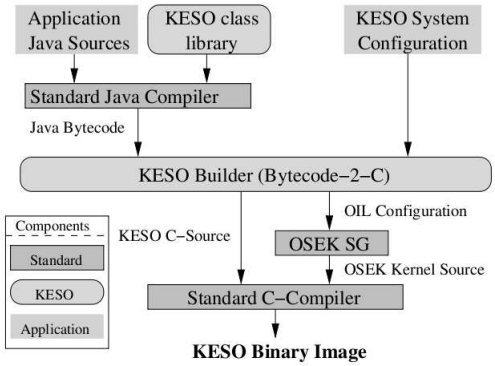

Fig. 2. System Generation Process [25].

KESO Builder also provides a Foreign Function Interface (FFI) for interfacing with C and C++ code. KESO FFI uses a static approach like Sun's KNI (see section 2), binding Java and native at compile time. A binding is created by extending KESO's *Wavelet* class and overwriting some of its methods. The class diagram in figure 3 illustrates how this is done. In the constructor of the *WaveletImplementation* class we specify the *join point*, which is the class that will be affected by the "weaving" (i.e. the binding). Overwriting the *ignoreMethodBody* to return *true*, we tell KESO not to generate code for methods of the "join point class". Overwriting the *affectMethod* method is possible to determine the C code that KESO should generate for each method of the "join point class". Finally, the new wavelet class (*WaveletImplementation*) must be registered in an appropriated class of the KESO compiler in order to be used during the bytecode to C translation stage.

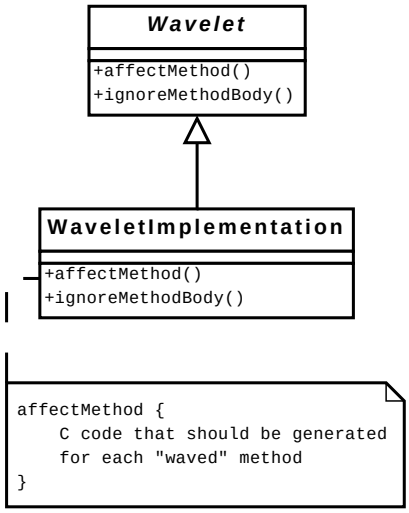

Fig. 3. Using KESO FFI.

The OSEK specification states that the operating system's components used by the application, such as tasks, alarms, and counters, are automatically generated from a configuration file. This configuration file is written using *OSEK Implementation Language* (OIL). KESO uses a similar configuration file written using *KESO Configuration Language* (KCL). As shown in figure 2, KESO Builder takes a KCL file and generates an OIL file from it. Then the OIL file can be used to generate the OSEK components needed by the application.

4 Interfacing EPOS components to Java

This proposal uses the KESO Foreign Function Interface to interface EPOS components to Java applications. KESO FFI is an interesting mechanism to extend Java since new functionalities can be added without changing the Java Virtual Machine itself.

We have used KESO FFI to create a binding for each EPOS mediator that should be accessed by Java, providing Java with hardware components. The approach used is shown in figure 4. The binding is performed by "FFI C gen code". "KESO C gen code" represents all code that is automatically generated by KESO compiler and does not belong to the binding. An example of hardware mediator that we brought to Java was the Universal Asynchronous Receiver Transmitter (UART), used for serial communications. In Java, the *put* method of

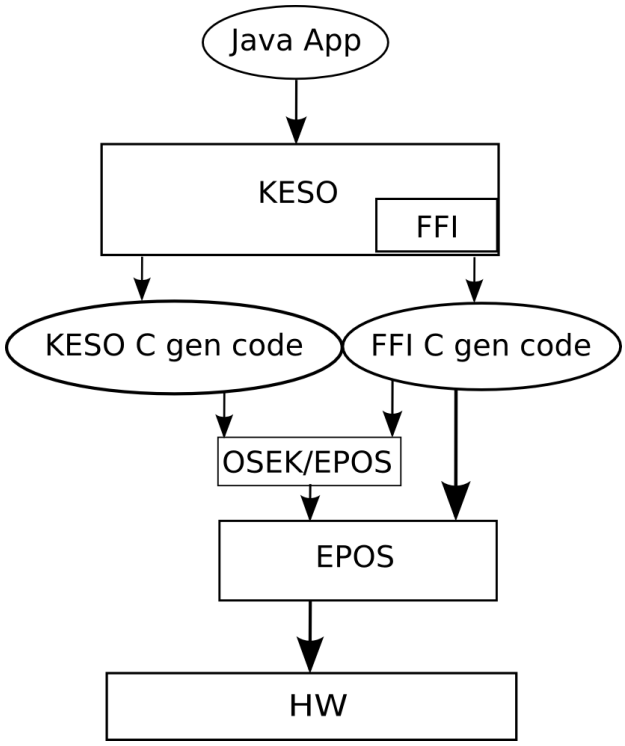

Fig. 4. Proposed approach.

the *UART* class has no implementation. Then, the *UART_Weavelet* class extends the *Weavelet* class and tells KESO which code should be generated by KESO FFI, i.e., the *UART_Weavelet* class plays the roles the *WaveletImplementation*

Java	EPOS
start	Thread::resume
sleep	Alarm::delay
join	Thread::join

Table 1. Implementing Java Thread class using KESO FFI.

class from figure 3. The binding code is basically a call to *put* method in the UART mediator in EPOS.

The binding code ("FFI C gen code") can run directly on EPOS, but the code generated by KESO ("KESO C gen code") needs an operating system that follows the OSEK standard. An OSEK/EPOS layer was recently developed to solve this problem [9]. Using this layer, EPOS behaves like any other OSEK operating system and can run Java applications generated by KESO. The OSEK/EPOS layer basically maps OSEK constructions in EPOS abstractions, respecting the OSEK semantics. For instance, OSEK tasks are mapped into EPOS Threads, and OSEK alarms into EPOS alarms.

Using KESO FFI we can go beyond the OSEK API, providing developers with Java classes of other APIs and implementing others specifications, such as the Java Standard Edition API and Java Real Time Specification. In order to demonstrate this possibility we have implemented the Java SE *Thread* class, using KESO FFI and EPOS system abstractions. Table 1 summarizes how we mapped Java Thread methods in EPOS methods.

We create a private method *_entry* in Java Thread class and we mapped it as the entry point of EPOS Threads. The *_entry* method calls the *run* method of Java Thread class which is empty by default and should be overwritten by classes that extend Java Thread. EPOS threads was created in *SUSPENDED* state to fulfill the semantic of Java Threads. So, *start* method of Java was implemented using the *resume* method of EPOS Thread class. To implement the static method *sleep* of Java, we used another system abstraction of EPOS, the *Alarm*.

5 Case study

In order to evaluate the proposal presented in section 4, we have developed a Java application that uses the UART mediator of EPOS to write characters on a serial device.

5.1 Description of the application

Figure 5 shows the Java application source code. Our application class *UART_SendSingleCharTest* extends the KESO class *Task*. This means that an instance of our class will be implemented by an OSEK task created by KESO, during the system generation. In turn, an OSEK task is implemented by the OSEK/EPOS layer as an EPOS thread. The *launch* method is the task's entry point.

```
package test;

import keso.core.Task;

public class UART_SendSingleCharTest extends Task {
    public void launch() {
        char message = 'H';

        while (true) {
            UART.put(message);
        }

    }

}
```

Fig. 5. Java UART application.

In Java, the *put* method of the *UART* class has no implementation. Then, the *UART_Weavelet* class extends the *Weavelet* class and tells KESO which code should be generated by KESO FFI, i.e., the *UART_Weavelet* class plays the roles the *WaveletImplementation* class from figure 3. The binding code is basically a call to *put* method in the UART mediator in EPOS.

The main classes used in the UART application are shown together in the class diagram in Figure 6. Classes in dark gray represent Java code and classes in light gray represent C and C++ source code. The *UART* class on the Java side is the Java class *UART* that has empty method bodies. The *UART* class on the C/C++ side represents the interface of the UART hardware mediator in EPOS and has implementations for several architectures and platforms.

Figure 7 shows the configuration file of the UART application written in *KESO Configuration Language* (KCL). The UART application is executed in a single processor (node), called "epos" and in a single domain (i.e. in a single JVM). This domain has only one task called "task1". "task1", is set to execute when the system initiates. "MainClass" parameter specifies which class implements "task1" which is *UART_SendSingleCharTest*. "MainMethod" parameter specifies that the *launch* method of *UART_SendSingleCharTest* class is the task's entry point.

5.2 Evaluation

We have compiled the UART application for IA32 and PPC32 architectures. We used GCC (gcc and g++) version 4.0.2 for both architectures [2]. In the IA32 architecture, we compiled for PC platform. In the PPC32 architecture, we compiled for ML310, a development board from Xilinx which contains a VirtexII-Pro XC2V30 FPGA (Field Programmable Gate Array) [7]. This FPGA has two IBM POWER PC 405 32-bit processor hardcores, but only one was used by our

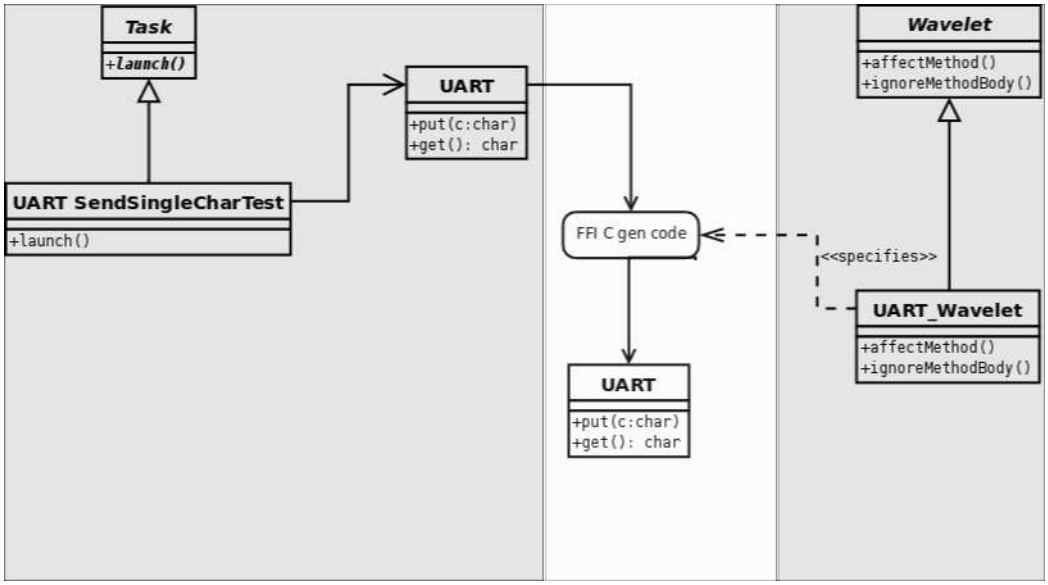

Fig. 6. Overall UART application.

application. We executed the IA32 application on QEMU PC emulator version 0.10.0 [1]. The PPC32 application was executed on ML403, another development board from Xilinx which contains a Virtex-4 XC4VFX12 FPGA [6]. This FPGA also has the IBM POWER PC 405, used by our application.

To estimate the memory overhead (total footprint) generated by our approach, we compared the Java application to an equivalent application written in C++ which directly uses the UART mediator of EPOS. Results for the IA32 and PPC32 architectures are shown, respectively, in tables 2 and 3. The values were taken from the binary image using the GNU size tool [2]. The binary image contains the application and the runtime environment. In the C++ application the runtime environment is just EPOS. In the Java application the runtime environment is EPOS and KESO JVM runtime support.

The image for the Java application is bigger than its C++ equivalent for both architectures (1.87 times for IA32 and 2.63 times for PPC32). All image sections have increased: code (text), initiated global and static local variables (data) and non-initiated global and local variables (bss). The section with the largest increase was the data section, due to global variables used in KESO runtime support. The binding for UART takes 92 bytes from the total image size in the Java application for IA32 architecture and 112 bytes for PPC32 architecture. It corresponds, respectively, to 0.29% and 0.34% of the total image size.

```
Node (epos) {
    Target = "epos";

    Modules = "debug: uart_send_single_char_test";

    OsekOS (KesoOS) {
        STATUS = "EXTENDED";
    }

    Domain (dom1) {
        Heap = IdleRoundRobin {
            HeapSize = 1024;
            SlotSize = 8;
            Group = "Default";
        }

        Task (task1) {
            MainClass="test/UART_SendSingleCharTest";
            MainMethod="launch()V";

            Autostart = true {
                Appmode = "OSDEFAULTAPPMODE";
            }

        }

    }

}
```

Fig. 7. Configuration file of UART application.

	C++ (byte)	Java (byte)	Overhead (times)
text	16148	28645	1.77
data	84	1180	14.05
bss	420	1264	3.01
total	16652	31089	1.87

Table 2. Application and runtime environment size - IA 32 architecture.

	C++ (byte)	Java (byte)	Overhead (times)
text	12140	30504	2.51
data	100	1198	11.98
bss	126	840	6.67
total	12366	32542	2.63

Table 3. Application and runtime environment size - PPC 32 architecture.

6 Discussion

Applications for embedded systems usually have the necessity to interact with several kinds of hardware devices such as sensors, actuators, transmitters, receivers, and timers.

Foreign Function Interface is the way adopted by Java to overcome language limitations and allows for direct memory and hardware devices access. However, as we showed in section 2, the main Java FFI solutions does not provide developers with a resource-efficient way to interfacing with hardware devices or impose design limitations to this interface.

In this paper, we have shown a way to interface hardware components with Java applications for embedded systems. This was achieved using the foreign function interface of KESO JVM and EPOS.

EPOS provides for a portable way to interfacing hardware devices with user applications. This is accomplished by using the concept of hardware mediators which sustain an interface contract between system abstractions and the machine.

KESO JVM compiles the bytecode of a Java application to C code and generates the parts of JVM needed by the application. The KESO FFI also uses this static approach generating C code specified by *Wavelet* classes. Then, the C codes generated by the KESO compiler and by KESO FFI are compiled together to native code using a standard C compiler.

The C code generated by KESO compiler needs an OSEK operating system to execute. Using an OSEK/EPOS layer, EPOS behaves like an OSEK OS allowing Java applications to run on it. However, using KESO FFI is possible go beyond OSEK API implementing other APIs and specifications. We have demonstrated this implementing the Java standard class Thread.

We have evaluated our approach implementing an UART application using JAVA and KESO and an equivalent application written in C++ using EPOS directly. The application footprint of 33KB, including all runtime support, is acceptable for many embedded systems.

References

1. About-qemu. [Online; accessed March 19, 2010].
2. Gcc, the gnu compiler collection. [Online; accessed March 24, 2010].
3. gcj: The gnu compiler for java. [Online; accessed March 18, 2010].
4. Jazelle - arm. [Online; accessed March 18, 2010].
5. Sjc: Small java compiler. [Online; accessed March 18, 2010].
6. Virtex-4 ml403 embedded platform. [Online; accessed March 19, 2010].
7. Xilinx ml310 documentation and tutorials. [Online; accessed March 19, 2010].
8. Osek vdx portal, 2008. [Online; accessed August 20, 2008].
9. Harald Bauer. Entwurf eines OSEK Adaption Layers für das Betriebssystem EPOS. Master's thesis, Friedrich-Alexander-Universität, Erlangen-Nurnberg, Deutschland, 2008.
10. Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, New York, NY, USA, 2006. ACM.
11. Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The jx operating system. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association.
12. OSEK group. OSEK/VDX operating system version 2.2.3, February 2005.
13. Juan A. Holgado-Terriza and Jaime Viúdez-Aivar. A flexible java framework for embedded systems. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 21–30, New York, NY, USA, 2009. ACM.
14. Stephan Korsholm and Philippe Jean. The java legacy interface. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 187–195, New York, NY, USA, 2007. ACM.
15. Sheng Liang. *The Java Native Interface - Programmer's Guide and Specification*. Addison-Wesley, 1999.
16. Harlan McGhan and Mike O'Connor. Picojava: A direct execution engine for java bytecode. *Computer*, 31(10):22–30, 1998.
17. Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.
18. Felipe Pompeo Pereira. Proposta de uma máquina virtual java orientada à aplicação, 2007.
19. Fauze Valério Polpeta and Antônio Augusto Fröhlich. Hardware mediators: a portability artifact for component-based systems. In *In Proceedings of the International Conference on Embedded and Ubiquitous Computing, volume 3207 of LNCS, Aizu,Japan*, pages 271–280. Springer, 2004.

20. Wolfgang Puffitsch and Martin Schoeberl. picojava-ii in an fpga. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 213–221, New York, NY, USA, 2007. ACM.
21. Martin Schoeberl. A java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, 2008.
22. Luciano Secchi. Ambiente de execução para aplicações escritas em java no sistema EPOS, 2004.
23. Michael Stilkerich, Christian Wawersich, Wolfgang Schröder-Preikschat, Andreas Gal, and Michael Franz. An OSEK/VDX API for Java. In ACM, editor, *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*, pages 13–17, New York, 2006.
24. Inc. Sun Microsystems. *K Native Interface (KNI)*. Sun Microsystems, Inc., 2002.
25. Christian Wawersich, Michael Stilkerich, and Wolfgang Schröder-Preikschat. An OSEK/VDX-based Multi-JVM for Automotive Appliances. In Springer Boston, editor, *Embedded System Design: Topics, Techniques and Trends*, IFIP International Federation for Information Processing, pages 85–96, Boston, 2007.