ABSTRACTING HARDWARE DEVICES TO EMBEDDED JAVA APPLICATIONS

Mateus Krepsky Ludwich
Laboratory for Software and Hardware Integration – LISHA
Federal University of Santa Catarina – UFSC
P.O.Box 476, 880400900 - Florianópolis - SC – Brazil
mateus@lisha.ufsc.br

Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration – LISHA
Federal University of Santa Catarina – UFSC
P.O.Box 476, 880400900 - Florianópolis - SC – Brazil
guto@lisha.ufsc.br

ABSTRACT

In this paper we introduce a method to interface hardware components with embedded Java applications. Access to hardware devices is an important requirement for embedded software since the embedded system must interact with the environment where it is inserted on. At the same time, the use of very high-level languages, such as Java, facilitates the development of embedded systems because they provide features such as object-orientation, automatic memory management, and memory protection. We have evaluated our method in terms of performance, portability, and memory footprint. We also have developed Java bindings for a component for motion estimation in H.264 video coding, demonstrating the usability of our approach in a real-world scenario.

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 highly desirable 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 fulfill all constraints of memory consumption, time, and energy, which could be present in such 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 [Liang 1999]. 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) [Fröhlich 2001] abstracts hardware devices as components which a well-defined interface. Using the Foreign Function Interface of KESO Java Virtual Machine (JVM) [Thomn et al. 2010] we create the binding between these components and their Java counterparts.

Our work focuses on application-driven, statically-configured, and deeply embedded systems. Which means, embedded systems designed to execute a single application, where all the resources the application will need are known at design time. With this in mind, we can optimize the system generation, for example

generating only the hardware components needed by the application. In this scenario features such as reflection and dynamic class loading can be also removed from the JVM.

The contributions of this paper include a method to support the development of class libraries which elements are abstractions to access hardware devices, and experimental results exploring the use of this method to develop Java bindings for a Motion Estimation component for H.264 video encoding.

The next sections of this paper are organized in this way: section 2 presents related works encompassing hardware/software interfacing for Java. We present our approach to interface hardware components with Java applications in section 3. Concepts of EPOS and KESO are presented as needed to support the understanding of our proposal. We evaluate our approach in section 4 in terms of performance, portability, and memory footprint. Section 5 presents a Motion Estimation component for H.264 encoding as an example of "real-world" application. Our final considerations are presented in section 6.

2. RELATED WORK

Our work encompass the issue of supporting direct hardware access from Java and to provide this support in a well structured way while keeping in mind all the requirements of the embedded systems scenario.

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 an issue to Java language. Foreign Function Interface (FFI) is the approach used by Java to overcome this limitation since it allows Java to use constructions, such as C/C++ pointers, to direct access hardware devices. FFIs have also been used by Java platforms 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 [Liang 1999], [Korsholm and Jean 2007].

Java Native Interface (JNI) is the main Java FFI, which is used in *Java Standard Edition* platform (JSE) [Liang 1999]. In JNI, the binding of native code is performed during runtime. Which means, the JVM searches and load into itself the implementation of methods marked as *native*. Usually the implementation of native methods is stored in a dynamic library. This search and loading mechanism 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 [Sun Microsystem 2002]. KNI does not dynamically load native methods into JVM, 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) from the 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 using KNI.

KESO FFI, used in this work, focuses on embedded systems. Like KNI, KESO FFI does not perform dynamically loading of native methods. 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.

3. INTERFACING HARDWARE DEVICES WITH VERY HIGH LEVEL LANGUAGES

One of the main issues of languages that aim to be used for embedded systems development is to access hardware devices. This is highly desirable, since embedded systems often use hardware devices to interact with the environment where they are inserted on. Our proposal to interface very high level languages with hardware devices is based on exporting these devices to the API of the language. The hardware devices to be exported have a well defined interface, using the *hardware mediator* concept of EPOS. We have used the Foreign Function Interface of KESO JVM in order to export EPOS mediators to Java. This section explains how we have abstracted hardware components to be used by embedded Java applications. The approach

utilization is exemplified by abstracting to Java a Universal Asynchronous Receiver Transmitter (UART) hardware device.

EPOS uses the concept of *hardware mediators* in order to abstract specificities from different hardware devices. *Hardware mediators* sustain an interface contract between system abstractions (e.g. threads) and the machine allowing for these abstractions machine-independence. [Polpeta and Fröhlich 2004]. The generation of the mediator implementation for a specific machine is performed at compile time. Using static meta-programming techniques and function's inlining is possible to dissolve mediators among the abstractions that use it, which avoids time overhead in the use of mediators.

KESO 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 code at compile time. Furthermore KESO FFI provides a powerful API for specifying wrappers generation. Adopting some concepts of Aspect-Oriented Programming (AOP) [Kiczales et al. 1997], using the KESO FFI API it is possible to "write" *point cuts* specifying the *join points* of a Java program (such as Java methods and classes) that will be affected by the given *advices*. The advice in this case, is the code that represents the native method implementation. The *aspects* that group together *point cuts* and *advices* are represented in KESO FFI API by a *Weavelet* abstract class. Extending the *Weavelet* class and implementing some of its methods it is possible to specify which Java classes and methods should be affected and which native code should be generated.

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 1. The Java class, which represents the Java counterpart for the hardware mediator been abstracted, specifies methods signatures but no method implementation (since they represent native methods). Then, a weavelet class of KESO FFI is used to specify the implementation for each native method. More specifically, the weavelet class specifies which methods of the Java class we would like to intercept (pointcuts) and the respective code that should be generated (advices). During the translation of Java bytecode into C, the KESO compiler "wove" the methods of the Java class with the advices specified by the weavelet, generating the *binding code* which performs the interface between the Java class and the EPOS hardware mediator.

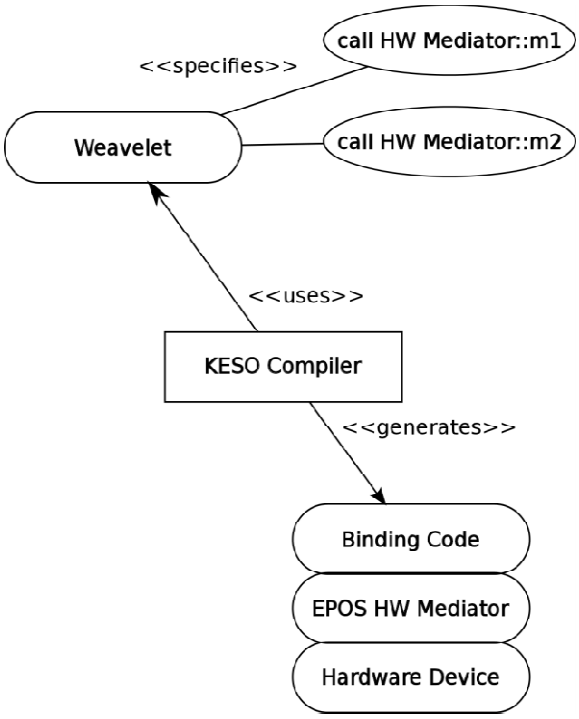

Figure 1. Accessing hardware devices.

The implementation of each native method, specified in the weavelet, is basically a call to each method of the EPOS mediator been interfaced. On the implementation of the method *affectMethod* (*Weavelet* interface) is

possible to specify a pattern to be matched which represents, for example, a method signature. It is also possible to specify which code should be generated when the KESO compiler finds the match specified by the pattern. Figure 2 shows the binding specification for a virtual method called *m1* which has one parameter of the type character and have no return (*void* type). The *eposHWMediator* is a field of the binding class which points to the actual hardware mediator of EPOS. Adding fields to binding classes is possible by implementing the *addFields* method of *Weavelet* interface.

```
public boolean affectMethod(IMClass clazz, IMMethod method, Coder coder)
throws CompileException {
  //  ...
  if  (method.termed("m1(C)V")) {
    coder.addln("obj0->eposHWMediator->m1(c1);");
    return true;
  }
  //  ...
}
```

Figure 2. Specifying the binding code.

EPOS objects allocated by a binding object, such as *eposHWMediator*, which is allocated when one calls *new* on the Java side are deallocated on the *finalizer* method of such Java class. As is possible to use deterministic algorithms for the KESO garbage collector there is a guarantee that finalizers are called.

The KESO FFI is integrated with the KESO compiler so, during the compilation of Java bytecode to C, instances of weavelets classes are created and used for generating the native code. Although the code specified by a weavelet is not subject of the static analysis performed by the KESO compiler, KESO FFI still presents some interesting advantages, which led us to use it. For example, if the KESO compiler identifies that the application code does not use some native method it does not generate the native code for that method, reducing the memory needs (which is high desirable in an embedded system scenario).

We wrote a small application using the UART hardware mediator to illustrate our proposal of abstracting hardware devices to embedded Java. The application, shown by figure 3, uses the UART to write characters on a serial device. The Java *UART* class is the Java counterpart abstraction for the UART hardware mediator of EPOS and has only native methods without any implementation. Using the approach shown by figure 2, we have created a weavelet class named *UART_Weavelet* which specifies the implementation for each *UART* method.

```
package test;
import keso.core.Task;

public class UART_Test extends Task {
  public void launch() {
    UART serial;
    serial  = new UART(19200, 8, 0, 1, 0);
    for(int  i  = 0;  i  < 10000; i++) {
      serial .put('M');
    }
  }
}
```

Figure 3. UART example.

4. EVALUATION

We evaluate our proposal of interfacing hardware devices with Java, described in section 3, in terms of performance, portability, and memory footprint. In ours experiments, we have used the UART example mentioned in section 3 and a component for Motion Estimation in H.264 video encoding.

The C/C++ code, generated by KESO compiler and by KESO FFI while using our approach, was compiled into native code using GCC (gcc and g++). The UART application was evaluated in IA32 and PowerPC32 (PPC32) architectures. The platform used for IA32 was the NANO-LX-800 board, a PC platform based on the Geode processor. The platform used for PPC32 was the ML310 from Xilinx which contains the Power PC 405 32-bit processor. The Motion Estimation Application was evaluated in the IA32 architecture, on a PC using the Intel Quad Core Q9550 processor.

This section describes the results for the UART mediator. Section 5 describes in details the Motion Estimation component as the obtained results for it.

4.1 Performance

A binding to interface very high level languages and hardware devices (i.e. FFI) should interferer as less as possible on the original response time of such devices otherwise its utilization can become impractical. In order to evaluate the proposal of section 3, we have measured the response time of the device been analyzed while using it directly (e.g. by a C++ application), and using it through ours Java bindings. We call *DeviceTime* the original response time which is composed by the time of the EPOS mediator plus the time of the physical device. The *TotalTime* adds to the *DeviceTime* the response time of the native method, including the call to the method and the method's return. With these concepts in mind, the time overhead generated by a FFI can be described by equation 1.

$$FFI_{Overhead(\%)} = \left(1 - \frac{DeviceTime}{TotalTime}\right) \times 100 \quad (1)$$

We have measured the total and the device time of the UART's put method. The method was called 10 thousand times in an application's execution, and the application was executed 30 times. We have used EPOS's *time stamp clock* to compute the time. The obtained overhead, according to equation 1 corresponds to less than 0.04% of the total execution time.

In order to estimate the relevance of the overhead value, we reproduce the UART experiment using Java Standard Edition which uses the Java Native Interface as FFI. We have used the *RXTX* (http://rxtx.qbang.org/) library which implements the *Java Communications API*, used for communicating with serial devices. The function *gettimeoftheday* was used to compute the device time, and the Java *System.nanoTime* method was used to compute the total time. Table 1 shows the obtained values as the values obtained while using our proposal.

The JNI based application presents an overhead of 1.5% which is about 38 times greater than the overhead obtained using your approach. Considering an application that must send a byte every 420μs and considering that a UART takes 417 μs to sent a byte (bound rate of 19200), an overhead of 1.5% (6.25 μs) would compromise the data transmission, leading to deadline misses in a real-time application.

Table 1. FFI time overhead.

FFI	Total (µs)	UART (µs)	FFI Overhead (%)
Proposal	517.74	517.54	0.04
JSE	8364683.74	8238695.07	1.5

4.2 Portability

Our Java bindings support to kinds of portability: platform portability, and software/hardware portability.

Since a Java binding developed using our approach rely on the concept of EPOS hardware mediators and the latter provides a machine-independent interface, the former can potentially exists in all architectures and machines for which EPOS has a port.

By software/hardware portability we meant that the same Java binding can be used either for a software or hardware implementation of the component been wrapped. This is possible due to the concept of *hybrid components* realized by EPOS, where the component preserves the same interfaces either in its software or hardware implementation [Marcondes and Fröhlich 2009].

4.3 Memory footprint

The binding code to wrap a hardware mediator should impact as less as possible on the code and data memory needed by the application to execute. In order to estimate the memory overhead generated by our approach, we have measured, using *GNU size*, the footprint of the binary image containing the whole system and the footprint specifically related for creating a Java binding.

Table 2 shows the obtained values for the UART example, which was developed for IA32 and Power PC32 architectures. The whole system size including the application, KESO JVM, and EPOS runtime has less than 33KB in both architectures, a suitable value for many embedded hardware platforms. The binding for UART takes 92 bytes from the total image size for IA32 architecture and 112 bytes for PPC32 architecture. It corresponds, respectively, to 0.29% and 0.34% of the total image size.

Table 2. Total footprint.

Section	IA32 (byte)	PPC32 (byte)
text	28645	30504
data	1180	1198
bss	1264	840
total	31089	32542

5. REAL-WORLD APPLICATION

In order to evaluate our proposal in a “real-world application” we have develop a Java component which computes Motion Estimation (ME) for H.264 video encoding. Motion Estimation is used to explore the similarity between neighboring frames in a video sequence, thus enabling them to be differentially encoded, improving the compress ratio of the generated bitstream [Wiegand et al. 2003]. ME is an significant stage for H.264 encoding, since it consumes around 90% of the total time of the encoding process [Li et al. 2004]. In order to improve the performance of ME, our component uses a data partitioning strategy where the motion estimation for each partition of the picture is performed in parallel by a *Worker* module which executed in a specific functional unit, such as a core of a multicore processor. There is also a *Coordinator* module, responsible to define the picture partition for each *Worker* and to provide them with pictures to be processed. The *Coordinator* is also responsible to gather results generated by *Workers* (motion cost and motion vectors) and to delivery these results back to the encoder. Figure 4 illustrates the interaction between *Coordinator* and *Workers* modules.

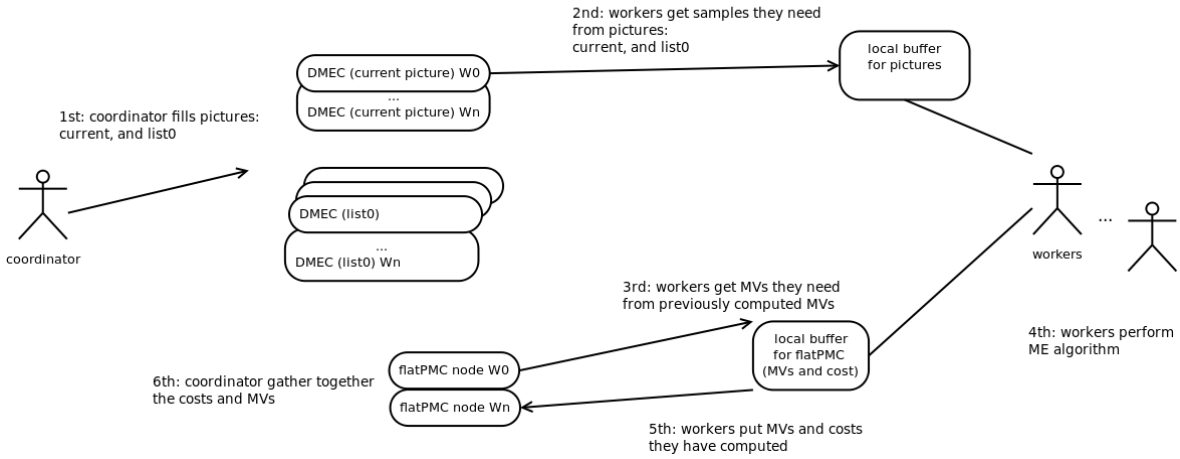

Figure 4. Interaction between Coordinator and Workers.

Our component is called *Distributed Motion Estimation Component* (DMEC), since the computation of ME is performed in parallel by *Workers* modules. However, this complexity is hidden from the Java application (e.g. H.264 encoder), which only sees a component for ME computation performed by a *match* method. The DMEC is implemented as a C++ component and it is exported to Java using the strategy presented at section 3, where is developed a Java binding for each object been abstracted.

Currently DMEC is implemented by software components, where *Coordinator* and *Workers* are threads running on distinct cores of a multicore processor. In spite of that, DMEC can be implemented by hardware components preserving the same interfaces available in the software version. We can achieve this by using the concept of EPOS hybrid components [Marcondes and Fröhlich 2009]. In that case our Java wrappers also remain the same. In a hardware implementation scenario *Coordinator* and *Workers* are IPs of a Multiprocessor System-on-Chip (MPSoC) and the communication between them is performed by the on-chip interconnection, such the ones described by [Javaid et al. 2010], [Popovici and Jerraya 2009].

We have written a Java application to use our DMEC component. From the ME point of view, the developed application plays the role of the H.264 encoder: it provides DMEC with the frames to be processed and it uses the results delivered by the component checking if they are correct. Figure 5 shows the main method of the application. The DMEC is used as a usual Java object.

```
public class DmecApp extends Task {
  public void launch() {
    DebugOut.println("DMEC APP is alive!");
    int width = 1920; int height = 1088; int maxRefPic = 1;
    PictureMotionEstimator pme = new PictureMotionEstimator(width, height, maxRefPic);

    Picture currentPicture = TestSupport.createPicture(width, height);
    Picture [] list0 = new Picture[2];
    for (int i = 0; i < list0.length; i++) {
          list0[i] = TestSupport.createPicture(width, height);
    }

    PictureMotionCounterpart pmc = pme.match(currentPicture, list0);

    TestSupport.testPMC(pmc, width, height, currentPicture, list0 );
    DebugOut.println("done: OK");
  }
}
```

Figure 5. DMEC Java application.

6. CONCLUSIONS

In this paper, we have introduced a method to interface hardware components with Java applications for embedded systems. Using the KESO FFI and EPOS we are able to abstract hardware components to Java while respecting constrains imposed by embedded systems.

We have evaluated our approach in terms of performance, portability, and memory usage. For an application using the UART hardware mediator the generated time overhead is less than 0.04% of the total execution time and our solution is 38 times faster than Sun's JNI. The memory footprint for such application was of 33KB, including all runtime support, which is suitable for many embedded systems.

Using EPOS we can achieve portability to several hardware platforms, and using the concept of hybrid components we can use the same Java bindings either for components implemented in hardware or software.

In order to evaluate our approach in a real-world application we have written Java bindings for a component which performs Motion Estimation for H.264 video encoding.

REFERENCES

Thomm, I. et al, 2010. Keso: an open-source multi-jvm for deeply embedded systems. *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, USA, pp 109–119.

Fröhlich, A. A., 2001. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Javaid, H. et al, 2010. Optimal synthesis of latency and throughput constrained pipelined mpsocs targeting streaming applications. *CODES/ISSS '10: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, USA, pp. 75–84.

Kiczales, G. et al, 1997. Aspect-oriented programming. *ECOOP*. SpringerVerlag.

Korsholm, S. and Jean, P., 2007. The java legacy interface. *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. New York, USA, pp. 187–195.

Li, X., Li, E., and Chen, Y.-K., 2004. Fast multi-frame motion estimation algorithm with adaptive search strategies in h.264. volume 3, pp. iii – 369–72 vol.3.

Liang, S., 1999. *The Java Native Interface - Programmer's Guide and Specification*. Addison-Wesley.

Marcondes, H. and Fröhlich, A. A., 2009. A Hybrid Hardware and Software Component Architecture for Embedded System Design. *International Embedded System Symposium*. Langenargen, Germany, pp. 259–270.

Polpeta, F. V. and Fröhlich, A. A., 2004. Hardware mediators: a portability artifact for component-based systems. *Proceedings of the International Conference on Embedded and Ubiquitous Computing, volume 3207 of LNCS, Aizu,Japan*, pp. 271–280.

Popovici, K. and Jerraya, A., 2009. Flexible and abstract communication and interconnect modeling for mpsoc. *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. Piscataway, USA, pp. 143–148.

Sun Microsystems, I., 2002. *K Native Interface (KNI)*. Sun Microsystems, Inc.

Wiegand, T. et al, 2003. Overview of the h.264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576.