Abstracting Hardware Devices to Embedded Java Applications

Mateus Krepsky Ludwich
Antônio Augusto Fröhlich

Laboratory for Software and Hardware Integration – LISHA
Department of Informatics and Statistics – INE
Federal University of Santa Catarina – UFSC
{mateus, guto}@lisha.ufsc.br

IADIS AC 2011

Introduction

- Why use very high-level languages (VHLLs), such as Java, for developing embedded systems?

- Key features for productivity improvement
  - Higher level of abstraction: object-orientation, automatic memory management
  - Robustness: memory protection
  - Facilitated debugging: exceptions, stack traces

- Several Java implementations already fulfill Embedded Systems (ES) requirements
  - memory, performance, real-time, energy
  - bytecode optimization, translation to native code, dedicated JVMs, generation techniques, specific memory models

The problem

- However, a useful implementation of Java for ES must provide a set of features to deal with the environment where the ES is inserted on

- These features are usually realized by hardware devices
  - Sensor, actuators
  - Transmitters, receivers
  - Timers

- This work proposes a method to abstract these hardware devices, and to provide them for the Java application developer, while matching ES requirements

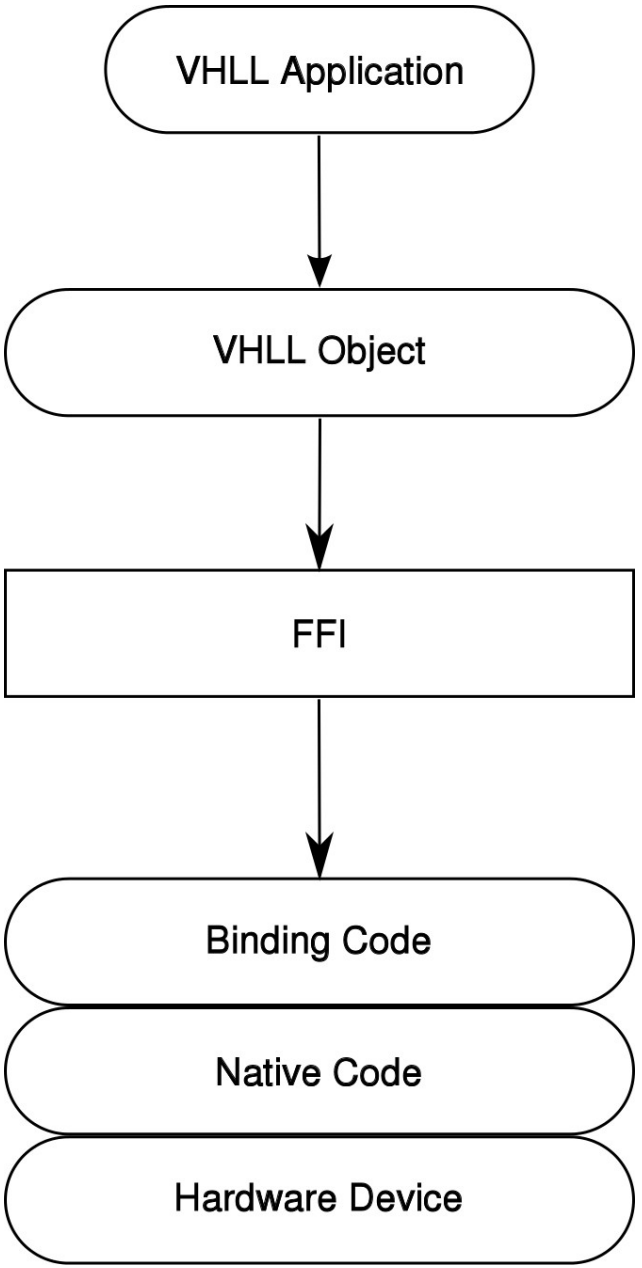
Interaction between Java and devices

- In order to control hardware devices, a programming language must be capable of writing in specific memory addresses (memory-mapped devices), and to use specific assembly instructions (devices controlled by dedicated I/O instructions)

- Java does not provide the concept of pointer, neither the concept of inline assembly...

- So, how Java deals with control of hardware devices?

Foreign Function Interface (FFI)



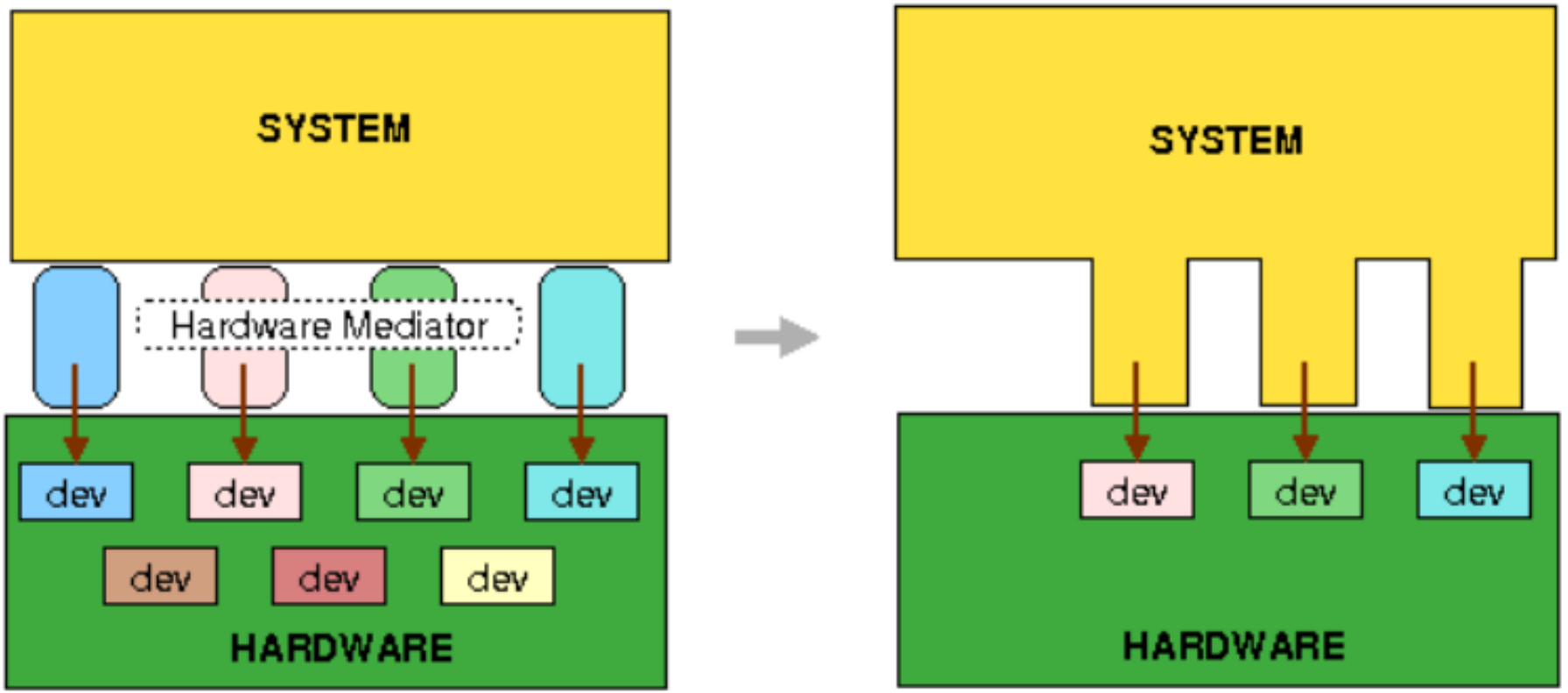
FFI Limitations

- However, an FFI itself does not guide the system developer on how to abstract hardware devices to Java
- It just provides a way to access constructions of another programming language

Proposal

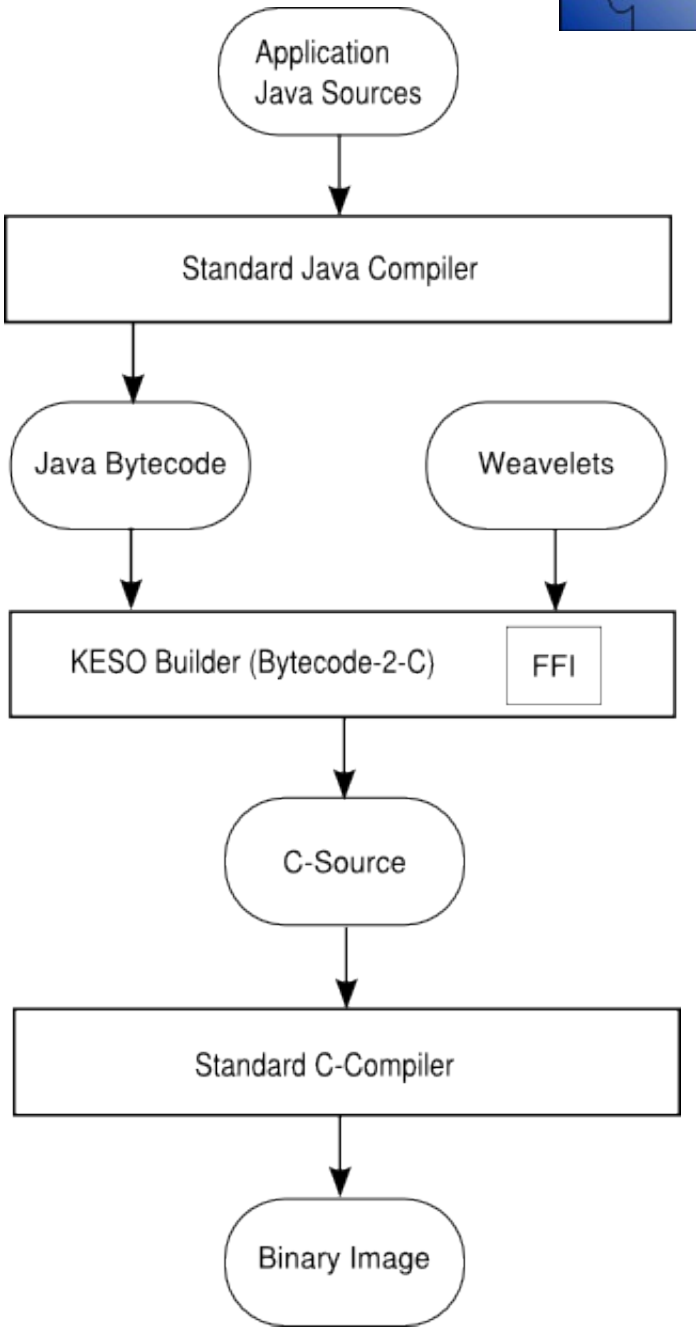
- To export hardware devices to the API of the language
- Hardware devices abstracted by hardware mediators
- Interface between hardware mediators and the Java language performed by a FFI and a JVM focused on embedded system (KESO JVM)

Hardware Mediators



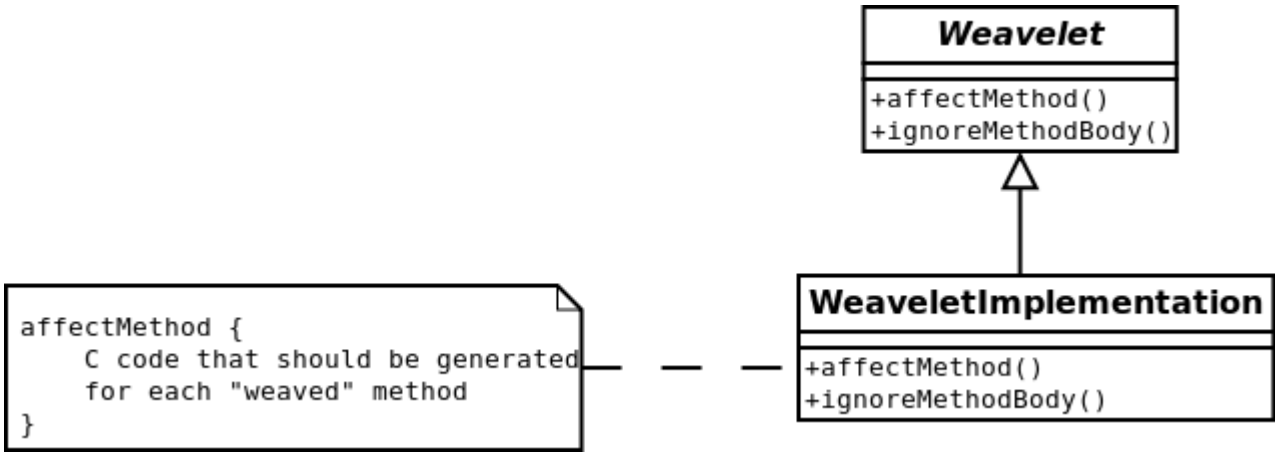
KESO Java Virtual Machine

- JVM focused on embedded systems
- Translates Java bytecode to C and generates the JVM runtime code
- It provides a FFI like JNI

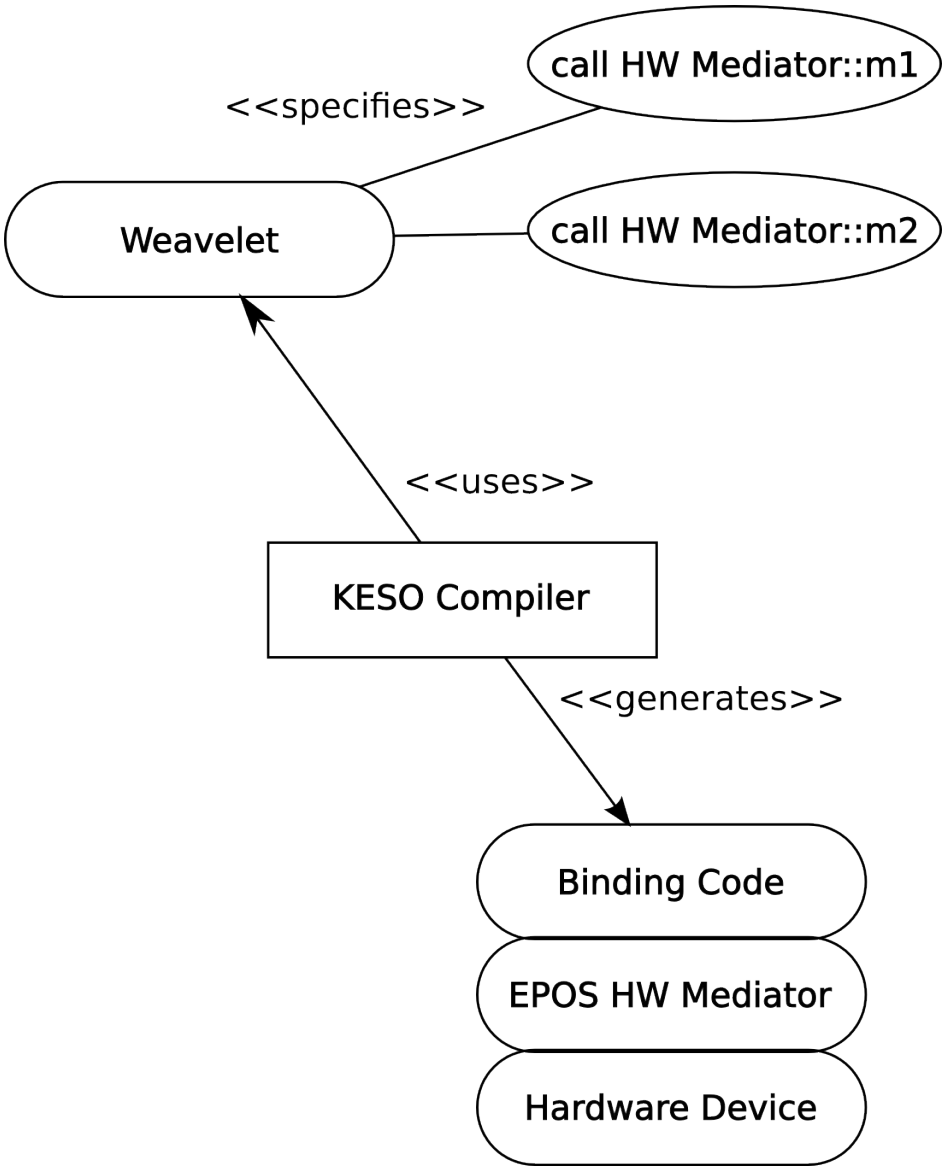


KESO FFI

- Uses a static approach like KESO itself
- Designed using concepts of AOP
  - The code that should be generated is specified by Weavelets classes that are used during the system generation



Approach


call HW Mediator::m1
<<specifies>>
Weavelet
call HW Mediator::m2
<<uses>>
KESO Compiler
<<generates>>
Binding Code
EPOS HW Mediator
Hardware Device

Specifying binding code


```
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;
  }
  // ...
}
```

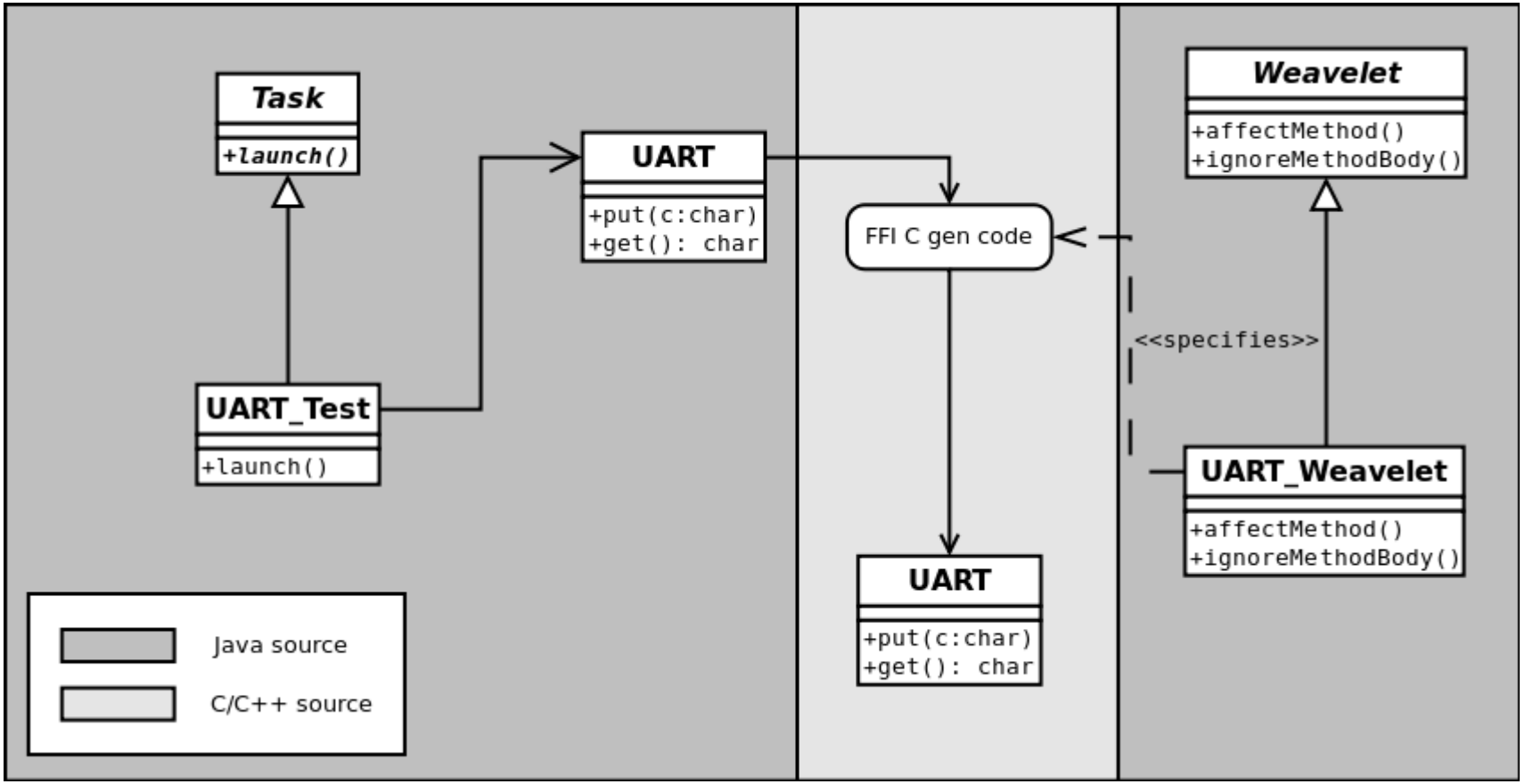
- C/C++ Code to be generated
- Method signature written in bytecode-like expression

Example application

```
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');
    }
  }
}
```

Application's architecture



Evaluation

- Two applications
  - UART example
  - Component for Motion Estimation in H.264 video encoding
- Architectures
  - IA32: NANO-LX-800 (UART), Intel Quad Core Q9550 (ME)
  - PowerPC32: PowerPC 405 (UART)
- Proposal evaluated in terms of performance, portability, and memory footprint

Performance

$$FFI_{overhead\,(\%)} = \left(1 - \frac{DeviceTime}{TotalTime}\right) x\,100$$

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

- Measured time of one specific native method: UART::put
- Proposal: KESO FFI + EPOS
- JSE: JNI + Linux
- Proposal is ~38x faster than the JNI based version
- Although proposal has a small overhead compared with direct UART access

Portability

- Platform portability
  - Binding code relies on hardware mediators
  - A same binding can be used without modifications in all platforms supported by EPOS
- Software/hardware portability
  - A same binding can be used either in a software or in a hardware implementation of the component been wrapped
  - Possible due to the concept of hybrid components realized by EPOS
    - A component preserve the same interfaces either in its software or hardware implementations

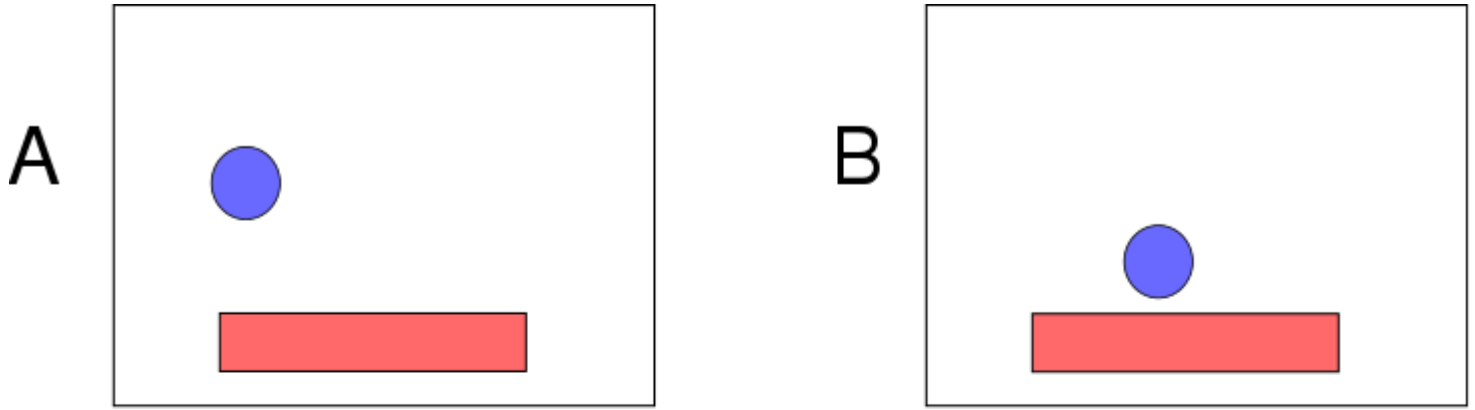
Memory footprint

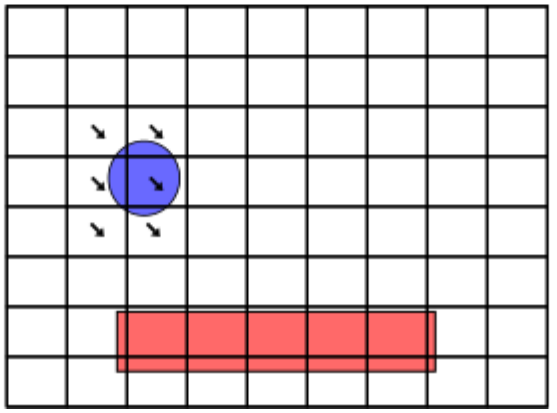
Section	IA32 (byte)	PPC32 (byte)
text	28645	30504
data	1180	1198
bss	1264	840
Total	31089	32542

- Whole system size <= 33 KBytes
  - Application + KESO JVM + EPOS
  - Binding code takes 92 bytes for IA32 (0.29%) and 112 bytes for PPC32 (0.34%) of the total system size

Real-world application

- Motion Estimation (ME) for H.264 video encoding





Real-world application

- ME is one of the main stages for H.264 video encoding
  - It takes ~90% of the total encoding time
- A multithreaded component computes ME in parallel

- All component 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

ME application code


```
public class DmecApp extends Task {
  public void launch() {
    // picture dimensions declaration...
    PME estimator = new PME(width,height,maxRefPic);

    Picture cp;
    Picture[] list0;
    // selecting pictures...

    PMC mvsc = estimator.match(cp, list0);
    // processing results
  }
}
```

Conclusions

- Introduced a method to interface hardware components and embedded Java applications
- Method evaluated in terms of performance, portability, and memory consumption
  - Time overhead is less than 0.04% of the total execution time
    - 38x faster than Oracle's JNI
  - Memory overhead is less than 0.3% of the total system size
  - Portability to several hardware platforms, and between hardware and software implementations of a component
- Approach scales from simple to more complex wrappers
  - UART and ME applications

Ongoing work

- Generalization of the proposal for others JVMs (e.g. NanoVM), and for others VHLLs (e.g. lua)
- Hardware mediators are the functional component of the device abstractions
- FFIs descriptions are handle in separated, and they are combined to hardware mediators to generate the final binding code

Abstracting Hardware Devices to Embedded Java Applications

Mateus Krepsky Ludwich
Antônio Augusto Fröhlich

Laboratory for Software and Hardware Integration – LISHA
Department of Informatics and Statistics – INE
Federal University of Santa Catarina – UFSC
{mateus, guto}@lisha.ufsc.br

IADIS AC 2011